AD-A240 534

ATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 01 Aug 1991 to 01 Jun 1993

4. TITLE AND SUBTITLE
TeleSoft, TeleGen2 Ada Host Development System, Version 4.1, for SPARCSystems, Sun-4/280 SPARC Processor (Hos & Target), 901128W1.11090

5. FUNDING NUMBERS

6. AUTHOR(S)
Wright-Patterson AFB, Dayton, OH
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB
Dayton, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-421.0891

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
TeleSoft, TeleGen2 Ada Host Development System, Version 4.1, for SPARCsystems, Wright-Patterson AFB, Sun Microsystems, Sun-4/280 Workstation (SPARC Processor) under Sun UNIX 4.2, Release 4.1 (Host & Target), ACVC 1.11.

91-11059

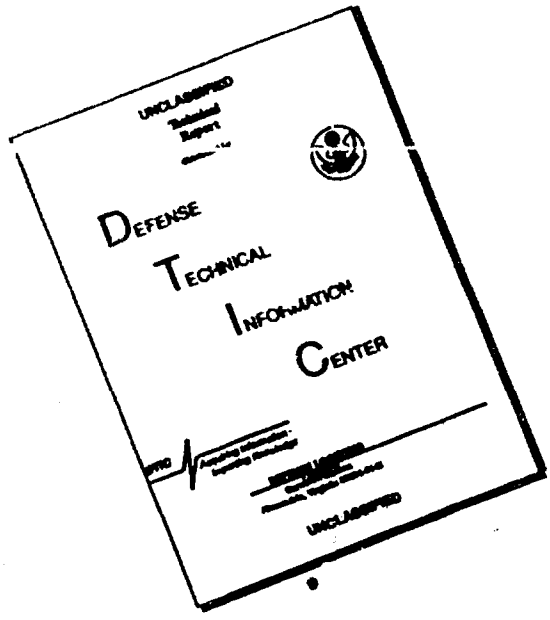
14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

DISCLAIMER NOTICE

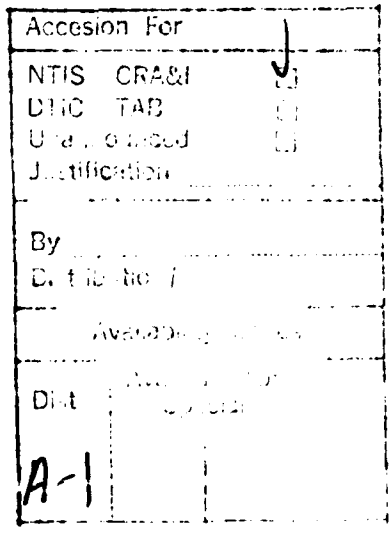


THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

AVF Control Number: AVF-VSR-421.0891
1 August 1991
90-06-28-TEL

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901128W1.11090
TeleSoft
TeleGen2 Ada Host Development System, Version 4.1,
for SPARCSystems
Sun-4/280 SPARC Processor => Sun-4/280 SPARC Processor

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright Patterson AFB OH 45433-6503



Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 28 November 1991.

Compiler Name and Version:	TeleGen2 Ada Host Development System, Version 4.1, for SPARC Systems
Host Computer System:	Sun Microsystems, Sun-4/280 Workstation (SPARC Processor) under Sun UNIX 4.2, Release 4.1
Target Computer System:	Sun Microsystems, Sun-4/280 Workstation (SPARC Processor) under Sun UNIX 4.2, Release 4.1

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901128W1.11090 is awarded to TeleSoft. This certificate expires on 1 March 1993.

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 28 November 1991.

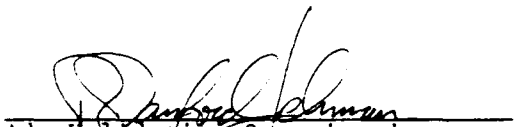
Compiler Name and Version:	TeleGen2 Ada Host Development System, Version 4.1, for SPARC Systems
Host Computer System:	Sun Microsystems, Sun-4/280 Workstation (SPARC Processor) under Sun UNIX 4.2, Release 4.1
Target Computer System:	Sun Microsystems, Sun-4/280 Workstation (SPARC Processor) under Sun UNIX 4.2, Release 4.1

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901128W1.11090 is awarded to TeleSoft. This certificate expires on 1 March 1993.

This report has been reviewed and is approved.

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503



Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

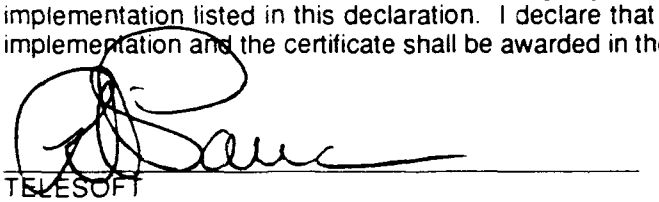
Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

Customer:	TeleSoft 5959 Cornerstone Court West San Diego CA 92121
Ada Validation Facility:	AVF, ASD/SCEL Wright-Patterson AFB, Ohio 45433-6503
ACVC Version:	1.11
Ada Implementation:	
Compiler Name and Version:	TeleGen2 Ada Host Development System, Version 4.1, for SPARCSystems
Host Computer System:	Sun-4/280 SPARC Processor
Target Computer System:	Sun-4/280 SPARC Processor

Customer's Declaration

I, the undersigned, representing TELESOFT, declare that TELESOFT has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration. I declare that TELESOFT is the OWNER of the above implementation and the certificate shall be awarded in the name of TELESOFT.



Date: 7 January 1991

TELESOFT
Raymond A. Parra, Director
Contracts/Legal

Revision A: 1/7/91

CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2 2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process or service of all requirements specified.

Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 12 October 1990.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026B
B85001L	C83026A	C83041A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

IMPLEMENTATION DEPENDENCIES

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L .Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35508I..J and C35508M..N (4 tests) include enumeration representation clauses for boolean types in which the specified values are other than (FALSE => 0, TRUE => 1); this implementation does not support a change in representation for boolean types. (See section 2.3.)

C35702A, C35713B, C45423B, B86001T, and C86006H check for the predefined type SHORT_FLOAT.

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P (4 tests) and C45532M..P (4 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45624A checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 5. For this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

C86001F recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete. For this implementation, the package TEXT_IO is dependent upon package SYSTEM.

B86001Y checks for a predefined fixed-point type other than DURATION.

CA2009C, CA2009F, BC3204C and BC3205D instantiate generic units before their bodies are compiled: this implementation creates a dependence on generic units as allowed by AI-00408 and AI-00530 such that the compilation of the generic unit bodies makes the instantiating units obsolete.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A840 use representation clauses specifying non-default sizes for access types.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method:

Test	File Operation	Mode	File Access Method
CE2102D	CREATE	IN_FILE	SEQUENTIAL_IO
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102I	CREATE	IN_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102E	CREATE	IN_FILE	TEXT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

The following 16 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE2107B..E CE2107G..H CE2107L CE2110B CE2110D
CE2111D CE2111H CE3111B CE3111D..E CE3114B
CE3115A

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

CE3304A checks that use_error is raised by a call to set_line_length or to set_page_length when the specified value is inappropriate for the external file. This implementation has no inappropriate values for either line length or page length.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 18 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests:

BA1001A	BA2001C	BA2001E	BA3006A	BA3006B
BA3007B	BA3008A	BA3008B	BA3013A	

C35508I..J and C35508M..N (4 tests) were graded inapplicable by Evaluation Modification as directed by the AVO. These tests attempt to change the representation of a boolean type. The AVO ruled that, in consideration of the particular nature of boolean types and the operations that are defined for the type and for arrays of the type, a change of representation need not be supported; the ARG will address this issue in Commentary AI-00564.

C52008B was graded passed by Test Modification as directed by the AVO. This test uses a record type with discriminants with defaults that has array components whose size depends on the value of a discriminant of type INTEGER. On elaboration of the type declaration, this implementation raises NUMERIC_ERROR as it attempts to calculate the maximum possible size for objects of the type. The AVO ruled that this behavior was acceptable, and that the test should be modified to constrain the subtype of the discriminants. Line 16 was modified to create a constrained subtype of INTEGER, and discriminant specifications in lines 17 and 25 were modified to use that subtype; these lines are given below:

```
16  SUBTYPE SUBINT IS INTEGER RANGE -128 .. 127;
17  TYPE REC1(D1,D2 : SUBINT) IS

25  TYPE REC2(D1,D2,D3,D4 : SUBINT := 0) IS
```

CD1009A, CD1009I, CD1C03A, CD2A21C, CD2A24A, and CD2A31A..C (3 tests) were graded as passed by Evaluation Modification as directed by the AVO. These tests use instantiations of the support procedure LENGTH_CHECK, which uses UNCHECKED_CONVERSION according to the interpretation given in AI-00590. The AVO ruled that this interpretation is not binding under ACVC 1.11; the tests are ruled to be passed if they produce Failed messages only from the instantiations of LENGTH_CHECK--i.e., the allowed Report.Failed messages have the general form:

" * CHECK ON REPRESENTATION FOR <TYPE_ID> FAILED."

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

TeleSoft Customer Support
5959 Cornerstone Court West
San Diego CA 92121-9891

For a point of contact for sales information about this Ada implementation system, see:

TeleSoft Sales
5959 Cornerstone Court West
San Diego CA 92121-9891

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3790
b) Total Number of Withdrawn Tests	81
c) Processed Inapplicable Tests	98
d) Non-Processed I/O Tests	0
e) Non-Processed Floating-Point Precision Tests	201
f) Total Number of Inapplicable Tests	299
g) Total Number of Tests for ACVC 1.11	4170

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 299 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

```
-O 2M

-L

-m
```

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	200
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"

MACRO PARAMETERS

```
$MAX_STRING_LITERAL    '"' & (1..V-2 => 'A') & '"'
```

The following table lists all of the other macro parameters and their respective values:

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_646
$DEFAULT_MEM_SIZE	2147483647
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	TELEGEN2
$DELTA_DOC	2#1.0#E-31
$ENTRY_ADDRESS	ENT_ADDRESS
$ENTRY_ADDRESS1	ENT_ADDRESS1
$ENTRY_ADDRESS2	ENT_ADDRESS2
$FIELD_LAST	1000
$FILE_TERMINATOR	''
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	131_073.0
$GREATER_THAN_FLOAT_BASE_LAST	3.40283E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	4.25354E+37

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        0.0

$HIGH_PRIORITY          63

$ILLEGAL_EXTERNAL_FILE_NAME1
                        "BADCHAR*^/%"

$ILLEGAL_EXTERNAL_FILE_NAME2
                        "/NONAME/DIRECTORY"

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        'PRAGMA INCLUDE ("A28006D1.TST");'

$INCLUDE_PRAGMA2        'PRAGMA INCLUDE ("B28006F1.TST");'

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483647

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     C

$LESS_THAN_DURATION     -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -131_073.0

$LINE_TERMINATOR        ASCII.LF

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        MCI'(OP => NOP);

$MACHINE_CODE_TYPE      Opcodes

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                -2147483648
```

MACRO PARAMETERS

$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	TELEGEN2
$NAME_SPECIFICATION1	/tmp/X2120A
$NAME_SPECIFICATION2	/tmp/X2120B
$NAME_SPECIFICATION3	/tmp/X3119A
$NEG_BASED_INT	16#FFFFFFFE#
$NEW_MEM_SIZE	2147483647
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	TELEGEN2
$PAGE_TERMINATOR	ASCII.FF
$RECORD_DEFINITION	TEST_WITHDRAWN
$RECORD_NAME	TEST_WITHDRAWN
$TASK_SIZE	32
$TASK_STORAGE_SIZE	2048
$TICK	0.02
$VARIABLE_ADDRESS	VAR_ADDRESS
$VARIABLE_ADDRESS1	VAR_ADDRESS1
$VARIABLE_ADDRESS2	VAR_ADDRESS2
$YOUR_PRAGMA	NO_SUPPRESS

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to linker documentation and not to this report.

SUN-4 Validation Information

This document presents information required for the validation of TeleGen2, version 4.01, for the SUN-4. It lists the options for the TeleGen2 Ada compiler and linker and includes annotations to the Ada Language Reference Manual (LRM), including Appendix F.

1. Compiler and Linker options

The Ada compiler is invoked via the *ada* command. The general syntax of the command is:

ada [<option>,...] <input_spec>

where: **<input_spec>** is one or more Ada source files and/or an input-list file (<name>.ilf) that lists the names of Ada source files. Note: ".ada" will be appended to a source file name if no suffix is present.

Option	Action	Default
Common options:		
-l(ibfile <libname>	Specify name of library file.	-l liblst.alb
-t(emplib <sublib>{,...}	Specify temporary list of sublibs.	None
-V(space_size <value>	Specify size of virtual space (Kbytes).	-V 2000
-v(erbose	Output progress messages.	(Opposite)
-L(ist	Generate *source/error listing.* (Cannot be used with *-F*).	(*Opposite*)
-F(ile_only_errs	Generate listing containing errors only. (Cannot be used with *-L*).	(Opposite)
-C(ontext <value>	Request <value> context lines with each error reported.	-C 1
-S(ource_asm	Generate source/assembly listing.	(Opposite)
-E(rror_abort <value>	Abort compilation after specified number of errors or warnings.	-E 999
-e(rrors_only	Run front end only (for error checks).	Full compile
-i(nhibit <key†>	Suppress checks and source information in object code.	(Opposite)
-d(ebug	Include debug information with object. (*-d* automatically sets *-k*.)	(Opposite)
-k(eep	Retain intermediate representation of unit. Must be used if *aar* or *aopt* is to be used on the unit.	(Opposite)
-m(ain <unit>	Produce executable code for <unit>.	(Opposite)
-O(ptimize <key†>	Optimize code.	(Opposite)
-u(pdate_lib <key>	<key> = s: update library after each source is processed; <key> = i: update after compiler invocation.	u s
-x(ecution_profile	Output profile code in object.	(Opposite)

The Ada linker is invoked via the *ald* command. The linker links object code to produce an executable module. The syntax of the command is:

ald [<option>,...] <unit>

where: **<option>** is none or more of the options in the table below.

<unit> is the name of the main program unit whose object code is to be bound and/or linked; <unit> is required.

Option	Action	Default
Common options:		
-l(ibfile <libname>	Specify name of library file.	-l liblst.alb
-t(emplib <sublib>{,...}	Specify temporary list of sublibs.	None
-V(space_size <value>	Specify size of virtual space (Kbytes)	-V 2000
-v(erbose	Output progress messages.	(Opposite)
-b(ind_only	Call the binder only: produce elaboration code and link script.	Linker too
-o(utput <file_spec>	Put executable code in <file>.	In *<unit>*
-p(ass_objects "<string>"	Pass "string" arguments directly to link editor.	(Opposite)
-P(ass_options "<string>"	Pass "string" options directly to link editor.	(Opposite)
-S("asm_listing"	Generate assembly listing for	(Opposite)
-T(raceback <value>	Set depth of exception traceback to <value> levels.	-T 15
-x(ecution_profile	Include profile information in executable module.	(Opposite)
Tasking options:		
-D(elay_NonPreempt	Specify non-preemptive delay.	*Pre*emptive
-X(ception_Show	Report unhandled exceptions in tasks.	(Opposite)
-w("timeslice" <value>	Limit task execution time to <value> msec.	-w 0
-Y <value>	Allocate <value> bytes (long) for task stack.	-Y 8192
-y <value>	Allocate <value> bytes (natural) for stack guard.	-y 1024

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

   ...
   type INTEGER is range -2_147_483_648 .. 2_147_483_647;
   type SHORT_INTEGER is range -32_768 .. 32_767;
   type SHORT_SHORT_INTEGER is range -128 .. 127;

   type FLOAT is digits 6 range -1.70141E+38 .. 1.70141E+38;
   type LONG_FLOAT is digits 15 range -8.98846567431158E+307
                                .. 8.98846567431158E+307;

   type DURATION is delta 6.10351562500000E-005 range -86400.0 .. 86400.0;
   ...

end STANDARD;
```

2. LRM Reference Information

This chapter provides information related to TeleGen2's implementation of the LRM and is divided into two major sections.

- Section 2.1, "Implementation," describes TeleGen2's implementation of tasking, exception handling, pragma Interface, machine code insertions, and interrupt handling.
- Section 2.2, "Annotations," describes the implementation-dependent portions of the Ada language for the TeleGen2 compiler and the run-time environment. The information is presented in the order in which it appears in the LRM. However, only those language features that either are not fully implemented by the current release of TeleGen2 or require further clarification are included.

2.1. Implementation

The following sections describe TeleGen2's implementation of tasking, exception handling, pragma Interface, machine code insertions, and interrupt handling.

2.1.1. Tasks

task stack — When a task is elaborated, a fixed size stack is allocated on the heap for its use. You can specify the size by using a length clause. This stack is deallocated and its storage is made available for other purposes when the task terminates.

task control block — A task control block (TCB) is allocated for each task and is deallocated when the master scope for the task is exited.

tasking support — The run-time requirements of Ada tasking are met principally by the machine-independent RSP component of the AEE. Three of the TDRSP packages, TD_Machine_State_Manager, TD_Delay_Manager, and TD_Interrupt_Manager, provide the target-dependent foundation for these facilities.

rendezvous — The RSP implements the semantics of the Ada rendezvous. Tasks calling entries in other tasks are queued on doubly-linked lists and left waiting until the called entry is free to serve the caller. Tasks performing entry calls that are not timed wait on a single list. The server or caller, whichever comes last, ensures that the server is active (or placed on the ready queue) so that the rendezvous code can be executed. A stack of callers is maintained during the rendezvous to allow for nested accept statements and proper relinquishing of the appropriate caller at the end of a rendezvous.

Parameter passing at a rendezvous depends on a parameter block built by the caller at the time of the call. A pointer to the block is passed to the called task when it reaches the rendezvous point.

delay expiration

After a specified delay has expired, an interrupt occurs in the task executing at the time of expiration. The task that executed the delay statement then becomes executable, and the executable task of highest priority is dispatched. If the interrupted task and the task whose delay has expired are of equal priority, the delayed task is dispatched.

2.1.2. Exceptions

handling exceptions

Each declared exception is identified by an address that points to the location where the exception name resides. When an exception is first raised, the address of the exception identifier and the address where the exception occurred are passed to the run-time system via various traps. The traps are encoded to minimize the code space and to provide the reason for the exception. The reraising of an exception (i.e., raising the handled exception inside the handler) passes along the same information along with the original raise address. The occurrence address is then used to determine the scope where the exception occurred. Next, the scopes are searched and the dynamic call link is followed until an exception handler is found for the raised exception.

exception tables

The Ada scopes with associated range of addresses and applicable exception handlers are encoded into a set of tables associated with each compilation unit. Therefore, there is no run-time overhead as each subprogram is entered, and the run-time costs are only incurred when an exception is raised. In designing the exception handling system, first priority is given to minimal space and time impact of exceptions on regular code when not raised. Second priority is given to reduced static space for exception tables. Following these constraints in importance are the goals of exception traceback quality and the speed of exception handling.

exceptions in tasks and non-Ada exceptions

TeleGen2 adheres strictly to the requirements of exception handling within tasks. See LRM Chapters 9 and 11. Hardware or operating system exceptions are converted into Ada exceptions and passed through the standard exception handlers. Foreign language exceptions may or may not be caught by the Ada exception handler. For this reason, foreign language routines should be tested separately and then added to Ada code.

classes of Ada exceptions

There are two classes of Ada exceptions: standard predefined exceptions and user-defined exceptions. The standard predefined exceptions are further divided into five classes: constraint_error, program_error, storage_error, numeric_error, and tasking_error. Additional information is also provided by the compiler and run-time system to distinguish between various cases of errors that raise the same predefined exception, e.g., between discriminant_check and access_check.

unhandled exceptions

When an exception is not handled by a user exception handler or by the termination of a task, the exception is called an unhandled exception (exceptions in tasks are not propagated outside the task except for special rendezvous conditions; refer to Chapter 9 of the LRM). Special reporting information is provided to the user in this case. A standard format message is sent to standard output, indicating the exception kind, exception reason, and the call chain for the exception propagation.

exception traceback

The exception traceback reports relevant information on inserted sections of code, due to macro-expanded generic instantiations or due to inline subprogram insertions. The original location of the exception raise point in the inserted code is indicated as well as the point of the insertion. If there are several levels of insertion (due to inlines containing inlined calls, etc.), only the innermost level and the outermost insertion point are reported.

The information from the last raise point is always complete as the stack is not unwound until a handler is located. The information from the original raise to the last reraise is stored in a special table internal to the run-time system. If that table overflows, a limited amount of information is lost in the call chain; however, this overflow is very unlikely. If another nested exception occurs from the handler activities and it is caught and handled, then the information on the raise-to-reraise call chain is also lost; however, this is a very rare situation.

2.1.2.1. Example of an unhandled exception

The following program raises the unhandled exception NUMERIC_ERROR.

```
 1  procedure Except is
 2    procedure proc is
 3
 4      procedure proc2 is
 5        i,j,k: integer;
 6      begin
 7        i := 0;
 8        k := j/i;
 9      end proc2;
10
11    begin
12        proc2;
13        exception
14          when NUMERIC_ERROR =>
15            raise;
16    end proc;
17
18  begin
19      proc;
20      exception
21        when NUMERIC_ERROR =>
22          raise;
23
24  end Except;
```

The exception traceback looks as follows:

```
>>> Unhandled exception: NUMERIC_ERROR (division by zero)
    raised in sec/except.proc2 at line 8
    called from sec/except.proc at line 12
    reraised in sec/except.proc at line 15
    called from sec/except.except at line 19
    reraised in sec/except.except at line 22
```

The original raise point is in unit "sec/except," procedure "proc2" at source line 8. Following the raise point are the various calls to "proc2" and the points of reraise (if any) as the call chain gets unwound. "proc2" was called from procedure "proc" at source line 12. Procedure "proc" also has an exception handler for the exception "NUMERIC_ERROR" that was raised, and that was reraised at source line 15. Normally the programmer might choose to place some exception recovery code or clean-up code here prior to reraising the exception. As the call chain gets further unwound, we see that "proc" itself was called from procedure "except" at source line 19. Again procedure "except" also had a handler for exception "NUMERIC_ERROR," and it also reraised the exception which is reflected in the last traceback line.

2.1.2.2. Procedures inserted into the source program

Occasionally, exceptions occur in procedures that are inserted into the source program, either via generic instantiations or via inlining, as in the following example. In these cases, only the innermost and outermost source line numbers are displayed. For example, suppose that "proc2" within procedure "Except" was inlined.

```
 1  procedure Except is
 2    procedure proc is
 3
 4      procedure proc2 is
 5        i,j,k: integer;
 6      begin
 7        i := 0;
 8        k := j/i;
 9      end proc2;
10      pragma inline (proc2);
11
12    begin
13        proc2;
14        exception
15          when NUMERIC_ERROR =>
16            raise;
17    end proc;
18
19  begin
20      proc;
21      exception
22        when NUMERIC_ERROR =>
23          raise;
24
25  end Except;
```

The exception traceback at run time differs from that in the previous example.

```
>>> Unhandled exception: NUMERIC_ERROR (division by zero)
    raised in sec/proc.proc2 at line 8
           in sec/except.proc at line 13
    reraised in sec/except.proc at line 16
    called from sec/except.except at line 20
    reraised in sec/except.except at line 23
```

Because of "pragma inline (proc2)" at source line 10, every time the compiler identifies a call to "proc2" in the program, it expands the body of "proc2" into the source program at that point. This changes the manner in which the exception traceback mechanism reports the traceback. The original exception is reported at source line 8 correctly, but as this occurred in an inlined procedure, the subsequent line, "in sec/except.proc at line 13," relays the information that this procedure was inlined in procedure "proc" at source line 13. The remainder of the traceback is the same as in the previous example.

2.1.2.3. Suppressing source location information option

The user can choose to suppress source location information through compile time switches. In this case, the exception traceback is still provided with the maximum information possible. If line numbers in the source are suppressed, then the location (i.e., address) is provided instead. If source names are suppressed, then the source subprogram name is represented by the address of the subprogram. The original version of procedure Except was compiled to suppress line number information in the code.

```
 1  procedure Except is
 2    procedure proc is
 3
 4      procedure proc2 is
 5        i,j,k: integer;
 6      begin
 7        i := 0;
 8        k := j/i;
 9      end proc2;
10
11    begin
12        proc2;
13        exception
14          when NUMERIC_ERROR =>
15            raise;
16    end proc;
17
18  begin
19      proc;
20      exception
21        when NUMERIC_ERROR =>
22          raise;
23
24  end Except;
25
```

The subsequent traceback displays addressing information instead of the source line numbers as in the previous examples:

```
>>> Unhandled exception: NUMERIC_ERROR (division by zero)
    raised in sec/except.proc2 at address 00002562
    called from sec/except.proc at address 00002532
    reraised in sec/except.proc at address 00002550
    called from sec/except.except at address 000024FC
    reraised in sec/except.except at address 0000251A
```

2.1.2.4. System.Report_Error

The user has the capability to display the call chain leading up to a handled exception. By calling "System.Report_Error," the normal exception traceback format is sent to standard output. After reporting exception traceback information up to the point of invocation of "System.Report_Error," the traceback mechanism continues to report calls all the way out of the program.

In the example below, the original procedure Except includes a context clause for package System, and System.Report_Error is called in Except after NUMERIC_ERROR instead of raising the exception again.

```
 1  with System;
 2
 3  procedure Except is
 4
 5
 6    procedure proc is
 7
 8      procedure proc2 is
 9        i,j,k: integer;
10      begin
11        i := 0;
12        k := j/i;
13      end proc2;
14
15    begin
16        proc2;
17        exception
18          when NUMERIC_ERROR =>
19            raise;
20    end proc;
21
22  begin
23      proc;
24      exception
25        when NUMERIC_ERROR =>
26          System.Report_Error;
27
28  end Except;
```

The exception traceback produced ends with the invocation of System.Report_Error because no raise statements or calls occur after that.

```
>>> Unhandled exception: NUMERIC_ERROR (division by zero)
    raised in sec/except.proc2 at line 12
    called from sec/except.proc at line 16
    reraised in sec/except.proc at line 19
    called from sec/except.except at line 23
    Report_Error invoked in sec/except.except at line 26
```

If an exception is raised in a non-Ada routine and the Ada exception handler detects it, the following message is output along with the address of the instruction that caused the exception to be raised.

```
unit name and proc name unknown
```

Because most non-Ada routines do not follow Ada calling conventions, the traceback may be incomplete, especially if one or all of the frame pointers have been corrupted.

2.1.2.5. Exceptions in tasks

The binder -X switch tells the run-time system to send to standard output, in the standard exception format, a report for an exception raised in a task. This occurs when an exception is raised in a task and is not handled by it, causing the task to terminate.

2.1.3. Pragma Interface

The TeleGen2 system supports interfacing to other languages (for example, assembly language), provided the interface meets appropriate conditions.

Conditions for using pragma Interface

- All communication between non-Ada routines and the Ada program must be achieved via parameters and function results.
- The Ada routine must be described with an Ada subprogram specification in the calling program. (LRM 13.9)
- Non-Ada routines must be specified with an appropriate "pragma Interface" directive in the calling program, using the syntax defined in the LRM (13.9):

```
pragma Interface (<language>, <subprogram_name>);
```

- The routines must be assembled or compiled by a language processor whose calling, data representation, and run-time conventions are compatible with the forms supported by TeleGen2, and whose object file format is acceptable to the UNIX linker (internal requirement).
- Pragma Interface can only be applied to subprograms for which users can provide bodies. That is, enumeration literals, attributes, predefined operators, and derived subprograms are not valid. (Ada Issues, AI-00306)
- The pragma is allowed to "stand for" several subprograms. However, the pragma will be satisfied only for those subprograms declared earlier in the same declarative part or package specification. (AI-00306)
- If the pragma is accepted and is applied to certain subprograms, it is illegal to provide a body for any of these subprograms. It is immaterial whether the pragma appears before or after the body. (AI-00306)
- If the subprogram named in the pragma was declared by a renaming declaration, the pragma applies to the denoted subprogram, but only if the denoted subprogram otherwise satisfies the requirements. (AI-00306)

TeleGen2 offers the following four forms of pragma Interface.

```
pragma Interface(assembly, <subprogram_name>);
pragma Interface(FORTRAN, <subprogram_name>);
pragma Interface(C, <subprogram_name>);
pragma Interface(Pascal, <subprogram_name>);
```

Ada-to-assembly calling conventions are discussed in Section 2.1.3.2. Ada-to-FORTRAN is discussed in Section 2.1.3.5. Ada-to-C conventions are discussed in Section 2.1.3.3. Ada-to-Pascal conventions are the same as Ada-to-C conventions.

If your program requires access to routines written in languages other than the preceding, you need to write assembly language routines that map between the calling conventions of those languages. You also need to examine the assembly language listings carefully for the Ada calls and the target language entry code to ensure that parameters are being passed and received correctly.

The following table summarizes the major points of each type of language interface currently supported.

Table 1. Language interface summary

	Ada→ASM	Ada→C	Ada→Pascal	Ada→FORTRAN
Parameter passing:	Right to left	Right to left	Right to left	Right to left
Calling conventions:	Section 2.1.3.2	Section 2.1.3.3	Section 2.1.3.4	Section 2.1.3.5
Objects are passed:	By value if 4 bytes or smaller; by reference otherwise. Floating point numbers are always by value.	Scalars by value; composites by reference.	Scalars by value; composites by reference.	Always by reference.
Names are in:	Uppercase	Lowercase	Lowercase	Lowercase

2.1.3.1. General interfacing considerations

The Ada name of the designated subprogram is referenced directly as a global symbol; it must resolve to an identical symbol defined at link time, presumably by the foreign language routine to be called. One implication of this fact is that the name of the foreign language routine must conform to Ada identifier rules (e.g., it must start with a letter; contain only letters, digits, or underscore characters; etc.). These name restrictions can be circumvented, however, by the use of the TeleGen2-defined pragma, Interface_Information. Refer to Section 2.2.8.1, "Implementation-defined pragmas," for further information.

2.1.3.2. Calling assembly routines from Ada

Parameter passing and calling conventions

The conventions for calling routines written in assembly language are basically the same as for a call to an Ada-implemented routine.

2.1.3.3. Calling C routines from Ada

TeleGen2 supports pragma Interface to routines written in C and other languages that adhere strictly to C interface conventions.

This form of pragma Interface has the following syntax.

```
pragma Interface (C, <Ada_subprogram_name>);
```

Pragma Interface to C is intended for interfacing to code generated by C language compilers, or by other non-Ada compilers that follow C conventions. Interfacing to standard UNIX library routines is discussed within the context of examples, in Section 2.1.3.3.

Parameter passing

Because the C programming language specifies the passing of arguments strictly by value, only *in* arguments may reliably be passed to C functions. Although a C routine can legally include an assignment to a formal parameter, it is not guaranteed that the assignment will result in an update to the stack copy of the parameter on exit from the routine, as required for Ada's model for *out* and *in out* parameters. To pass values back to the calling program, you can specify that the value is returned through the function return mechanism. This permits the return of 2- and 4-byte non-aggregate data types. If you wish to return objects of other types, you can pass pointers to the objects in the calling program (arrays, etc.) where the results are to be stored. The called routine can then access these objects through the C pointer mechanism.

Table 2 shows Ada types and their corresponding C types.

Table 2. Type comparisons: Ada and C

Ada			C		
Type	Passed by	Bytes	Type	Passed by	Bytes
character	value	1	char	value	1
boolean	value	1	short	value	2
short_integer	value	2	short	value	2
integer	value	4	int	value	4
float	value	4	float	value	4
long_float	value	8	double	value	8
access	value	4	*	value	4
array	reference ‡		[]	reference	
record	reference ‡		struct	reference	
string			array of chars		

String parameters

In practice, one of the most commonly passed objects is the string. (Note that in the C language, strings are "character arrays;" there is no "string passing" as such in C or between C and other languages.)

C and Ada handle strings differently; it is not feasible for the Ada compiler to attempt an automatic translation between string-handling models when calling C routines from Ada. You must therefore be aware of the differences, and tailor your code to take explicit account of them.

Table 3. Ada and C string parameters

Ada String	C "String"
Passed by descriptor.	Passed by the address of the first element.
Not null terminated, but you can null-terminate explicitly.	Null terminated.
Carries implicit index values for first and last element.	Carries no implicit index values or lengths. The code handling the string must test for the null character that terminates the string.

‡ By reference if more than 4 bytes, by value if 4 bytes or less.

There are two ways to accommodate these differences, either by writing C code for handling Ada strings or vice versa.

Make C follow Ada conventions. If an Ada string is passed to a C routine, the C routine should be declared with three explicit arguments corresponding to the Ada string parameter. The first will be the address of the first element of the string, the second will be the index value associated with it, and the third will be the index value associated with the last element of the string. A null string in Ada is indicated by the index value of the last element being less than the index value of the first element.

Make Ada follow C conventions. The more common (and efficient) way to pass strings between Ada and C routines is for the Ada code to explicitly follow C conventions. To pass a string to a C routine, the Ada code would store a null terminator at the end of the string and pass the address of its first element. A C function returning a string would be declared as an Ada function returning a value of type System.Address.

One way a string may be "passed" between C and Ada is shown in Section 2.1.3.3.1.2, that follows.

The command line and the global environment

The following global names have been declared in the run-time code specifically for use in C programs:

```
extern int ada__argc_save;     /* Number of command line arguments*/
extern char **ada__argv_save; /* Pointer to array of argument strings*/
extern char **environ;          /* Pointer to the environment variable string*/
```

(Note that there are two underscores following "ada" in the first two global names.) These three variables are used in the same way as are the main program arguments *argc*, *argv*, and *envp*. (See your UNIX User's Guide for details.) Command line arguments such as these are accessed via pragma Interface to C. An example of accessing UNIX command line arguments is provided in the following, in Section 2.1.3.3.1.2.

Usage examples

This section has examples that explain the use of pragma Interface to C.

2.1.3.3.1.1. Random number example

This section is the first of two that show how pragma Interface to C may be used in Ada applications. In this section, an Ada procedure, Random_Number, is provided as an example. Random_Number generates and prints a random long

integer based on a user-entered seed value. It generates this number by making a direct call to the C library functions *srand* and *rand*. No user-written C code is required.

```
-------------------------------------------------------
          -- Ada procedure Random_Number --
-------------------------------------------------------

with Text_IO;
with Integer_Text_IO;
procedure Random_Number is

  Number : Integer;

  function srand (Seed : Integer) return Integer;
  pragma Interface (C, srand);

  function rand return Integer;
  pragma Interface (C, rand);

begin

  Text_IO.Put ("Enter seed for random number generator: ");
  Integer_Text_IO.Get (Number);
  Text_IO.New_Line;
  Number := srand (Number);

  Number := rand;
  Text_IO.Put ("Random number is: ");
  Integer_Text_IO.Put (Number);
  Text_IO.New_Line;

end Random_Number;
```

In the declarative part of the program, the specifications for *rand* and *srand* are defined in the usual fashion. The pragmas immediately follow the specifications. For information on the Ada rules for preparing pragma Interface specifications, refer to Section 13.9 of the LRM.

To make the Ada procedure Random_Number executable, compile and link *random.ada* as shown in the following. Note that file *random.ada* contains the main program Random_Number and that a copy of *random.ada* is in the product examples directory. During linking, the linker searches the C library for *srand* and *rand* routines.

```
ada  -v  -m random_number  $____/examples/random.ada
```

2.1.3.3.1.2. Command line argument example

This section provides another example of the use of pragma Interface to C. This time, an Ada procedure, Show_Argument, calls the C procedure Get_Argument to return command line arguments. The C procedure uses global variables *ada__argc_save* and *ada__argv_save*, which were mentioned in the preceding, "Command Line and Global Environment."

The text of the calling Ada procedure is given in the following; a copy of the procedure in also available in file *show_args.ada* in the product examples directory. Note the use of C pointer types and the Ada System.Address type to give the C routine access to the string object that is to contain the returned argument.

```
-------------------------------------------------------
          -- Ada procedure Show_Argument --
-------------------------------------------------------
with System;
with Text_IO;
procedure Show_Argument is

  package IIO is new Text_IO.Integer_IO (Integer);

  Position  :  Integer;
  Argument  :  String (1 .. 1000);
  Arg_Len   :  Integer;

  function get_argument (Parameter_1  :  Integer;
                         Parameter_2  :  System.Address)
                         return          Integer;

  pragma Interface  (C, get_argument);

begin

  Text_IO.Put ("Enter position number of argument: ");
  IIO.Get(Position);
  Arg_Len := get_argument (Position, Argument'Address);
  Text_IO.Put_Line ("Argument is: " & Argument (1 .. Arg_Len));

end Show_Argument;
```

The following is the text of the C routine called from the Ada procedure Show_Argument.

```
/* ------------------------------------------------------------------- */
/*                -- C routine get_argument --                         */
/* ------------------------------------------------------------------- */

int get_argument (position, arg_ptr)

int  position; /* position number of argument to be returned */
char  *arg_ptr;  /* pointer to string in which to store argument */

{
    extern int ada__argc_save;     /* number of command line arguments */
    extern char **ada__argv_save; /* pointers to command line arguments */

    int strndx;  /* loop counter/string index */
    char  c;       /* temporary character */

    /* check argument position number */
    if (position > ada__argc_save - 1) return (0);

    /* one pass for every character in the parameter */
    /* until the null character at the end of the    */
    /* parameter is found                            */
    for (strndx = 0 ; c = ada__argv_save[position][strndx] ; strndx++)
            arg_ptr[strndx] = c;

    return (strndx); /* return the length of the string */
}
```

The steps involved in making this Ada procedure executable are outlined in the following.

1. Compile the C routine To do this, compile Get_Argument with the native C compiler. For example

   cc -c $____/examples/get_arg.c

   where "cc" invokes the C compiler and the *-c* option tells the compiler to compile the source code in file *get_arg.c* without linking it. The resulting object code is stored in the file *get_arg.o*. Note that the command to invoke the C compiler may be different for your system.

2. Compile and link the calling Ada procedure First invoke the Ada compiler and then the Ada linker. For example

```
ada  -v  $____/examples/show_arg.ada

ald  -v  -p 'get_arg.o'   show_argument
```

where *show_arg.ada* is the source file containing the Ada procedure Show_Argument, *show_argument* is the executable file the linker produces and puts in the current working directory, and *get_arg.o* is the argument of the -p(ass_objects option. The *-p* option directs the compiler to include the object file *get_arg.o* in the link. The *-p* option can also appear on the command line for *ada* when the *-m* option is used. In this case, the compiler passes the option to the linker.

2.1.3.4. Calling Pascal routines from Ada

TeleGen2 supports pragma Interface to routines written in Pascal. This form of pragma Interface has the following syntax.

```
pragma Interface (Pascal,  <Ada_subprogram_name>);
```

Pragma Interface to Pascal is intended for interfacing to code generated by Pascal language compilers.

Parameter passing

Because the Pascal programming language specifies the passing of arguments strictly by value, only *in* arguments may reliably be passed to Pascal functions. Although a Pascal routine can legally include an assignment to a formal parameter, it is not guaranteed that the assignment will result in an update to the stack copy of the parameter on exit from the routine, as required for Ada's model for *out* and *in out* parameters. To pass values back to the calling program, you can specify that the value is returned through the function return mechanism. This permits the return of 2- and 4-byte non-aggregate data types. If you wish to return objects of other types, you can pass pointers to the objects in the calling program (arrays, etc.) where the results are to be stored. The called routine can then access these objects through the Pascal pointer mechanism.

Table 4 shows Ada types and their corresponding Pascal types.

Table 4. Type comparisons: Ada and Pascal

Ada			Pascal		
Type	Passed by	Bytes	Type	Passed by	Bytes
character	value	1	character	value	1
boolean	value	1	boolean	value	2
short_integer	value	2			
integer	value	4	integer	value	4
float	value	4	real	value	4
long_float	value	8			
access	value	4	^	value	4
array	reference ‡		[ ]	reference	
record	reference ‡		record	reference	
string			array of characters		

‡ By reference if more than 4 bytes, by value if 4 bytes or less.

2.1.3.5. Calling FORTRAN 77 routines from Ada

This section describes how to call FORTRAN 77 routines from Ada programs to be compiled with the TeleGen2 system. The techniques described here are primarily a combination of those described in the preceding on using pragma Interface directives to interface to other languages.

The next section describes how to express the declaration of a FORTRAN 77 subprogram in Ada source code, by way of the pragma Interface directive. The section after that describes the procedures that must be followed to pass parameters to FORTRAN 77 subprograms.

Note: The examples included are only intended to be representative, especially when passing characters. Please consult your FORTRAN user documentation for specific aspects of your implementation.

Ada declaration of FORTRAN 77 subprograms

Each FORTRAN 77 subroutine or function (referred to hereafter as a *FORTRAN subprogram*) that is to be called directly from Ada must be declared in a declarative part as an Ada procedure or function, respectively. The Ada declaration must be followed by a pragma Interface (FORTRAN) directive that specifies the name of the subprogram. Pragma Interface_Information supplies the mapping between the Ada name and the FORTRAN name of the subprogram. For example, the FORTRAN subprogram

```
SUBROUTINE ZERO
```

would have the Ada declaration

```
procedure Zero;
pragma Interface (FORTRAN, Zero);
pragma Interface_Information (Zero, "ZERO")
```

In the last line, "Zero" is the Ada name, and "ZERO" is the FORTRAN name.

The number of formal parameters in the FORTRAN subprogram and corresponding Ada subprogram declarations must be the same. However, because the FORTRAN language passes all parameters by reference (allowing the called routine to modify the referenced object), all parameters to FORTRAN routines should be declared in the Ada declaration as *in out* parameters. Parameters to FORTRAN routines are not copied in and out, as are Ada *in out* parameters, but, instead, are passed by reference. For example, the FORTRAN subprogram

```
SUBROUTINE THREE (A, B, C)
INTEGER*2 A
INTEGER*4 B
```

```
REAL*4     C
```

would have the following Ada declaration

```
procedure Three (A : in out short_integer;
                 B : in out integer;
                 C : in out float);
pragma Interface (FORTRAN, Three);
pragma Interface_Information (Three,"THREE");
```

Parameter passing is described in detail in the next section.

Passing parameters to FORTRAN 77 subprograms

Parameters to be passed to a FORTRAN subprogram must be variables or expressions of either a scalar type (that is, an integer type, a real type, or an enumeration type) or an array type whose elements are scalars. The one exception to this rule is that a simple record containing two Float or Long_Float values may be passed for FORTRAN complex types. Passing of other record types, access types, and task types is not supported. Table 5 shows a comparison of Ada and FORTRAN types. There are limitations on the FORTRAN types, as discussed below.

Table 5. Type comparisons: Ada and FORTRAN

Ada			FORTRAN		
Type	Passed by	Bytes	Type	Passed by	Bytes
character	value	1	character*1	reference	1
boolean	value	1	logical*2	reference	1
short_integer	value	2	integer*2	reference	2
integer	value	4	integer*4	reference	4
float	value	4	real*4	reference	4
long_float	value	8	real*8	reference	8
access	value	4			
array	reference ‡				
record	reference ‡				
[none]			real*16	reference	16
short_short_integer			byte	reference	1
2-float record			complex*8	reference	8
2-long_float record			complex*16	reference	16
string					

The TeleGen2 system, when calling FORTRAN routines, passes all parameters by reference automatically. That is, it passes the address of the argument, not its

value. When passing constants or expressions to FORTRAN, the TeleGen2 system stores the value in a temporary location and passes the address of the temporary location to the FORTRAN routine.

For example, a call to subprogram Three in the previous example, passing scalar variables Zero, One, and Two, would be

```
Three (Zero, One, Two);
```

Passing scalar variables

The type of a scalar variable passed to a FORTRAN subprogram must be compatible with the type of the corresponding formal parameter declared in that FORTRAN subprogram. The type comparison table (Table 5) shows the correspondence between FORTRAN and Ada scalar types. The FORTRAN types INTEGER*2, INTEGER*4, REAL*4, REAL*8 and LOGICAL*2 types correspond directly to the Ada types Short_Integer, Integer, Float, Long_Float, and Boolean respectively. The remaining FORTRAN types, CHARACTER, COMPLEX, and DOUBLE COMPLEX, may be used if the special procedures described below are followed.

FORTRAN supports implicit variable declarations and declarations of variables without explicit sizes (e.g., INTEGER vs. INTEGER*2 or INTEGER*4). However, these types can be overridden by compile-time switches. To avoid unexpected behavior due to the use of these switches, declare the formal parameters in the FORTRAN program with an explicit length.

Passing CHARACTER types

The FORTRAN 77 CHARACTER type may be represented in Ada by an access to a string parameter. For every argument of type CHARACTER, an extra argument is passed giving the length of the value. For example, a FORTRAN subroutine defined as

```
SUBPROGRAM PROCESS (LINE)
CHARACTER*80 LINE
   . . .
END
```

could be called as shown in the following program fragment.

```
subtype String80 is string(1..80);
type acc_String80 is access String80;

-- Define the FORTRAN routine
procedure Process (Line   : in out acc_String80);
pragma Interface (FORTRAN, Process);
pragma Interface_Information (Process, "PROCESS");

-- Sample calling sequence
procedure Caller is
  Str : acc_String80 := new String80;
  Len : Integer;
begin
   ...
   Str(1..10) := "California";
   Len := 10;
   Process (Str, Len);
   ...
end;
```

It is important to note that all of the string length parameters must be at the end of the parameter list. For example, for the following FORTRAN declaration

```
SUBROUTINE FTN (LINE1, LINE2)
CHARACTER**80 LINE1, LINE2
 . . .
END
```

the Ada declaration would be

```
procedure FTN (Line1, Line2 : in out acc_String80;
               len1, len2   : in out Integer);
```

Passing COMPLEX and DOUBLE COMPLEX types

The FORTRAN 77 COMPLEX and DOUBLE COMPLEX types can be represented in Ada by a simple record containing two fields of type Float or Long_Float, respectively. For example, a FORTRAN subroutine defined as

```
FUNCTION SQUARE (A)
DOUBLE COMPLEX SQUARE
DOUBLE COMPLEX A
SQUARE = A * A
RETURN
END
```

could be called as shown in the following program fragment.

```
-- Define a FORTRAN Double Complex data type
type Double_Complex is record
  Real      : long_float;
  Imaginary : long_float;
end;

-- Define the FORTRAN routine
function Square (C : in Double_Complex);
Pragma Interface (FORTRAN, Square);

-- Sample calling sequence
procedure Caller is
  x,y: Double_Complex;
begin
   ...
   x := ( 1.0, -2.5 );
   y := Square (x);
   ...
end;
```

2.1.3.6. Calling Ada from other languages

Although the Ada LRM provides pragma Interface for calling routines written in other languages from Ada programs, it makes no provision whatsoever for calling Ada subprograms from code written in other languages. There are good reasons for that. An Ada program carries an implicit context that is not visible outside of the Ada run-time system, and it would be impossible, in practical terms, for the LRM to require implementations to support such usage in a general fashion.

Nevertheless, there are real cases where the ability to call Ada code from other languages is at least useful, if not an absolute requirement for an application. A common example would be an application built around an existing utility package, where the package has a programmatic interface in which users are expected to pass in the addresses of specific event-handling routines that they have coded. If appropriate restrictions are observed, it is possible to use the TeleGen2 system in this manner. You should be aware, however, that any such usage is outside the scope of the Ada LRM, and is not likely to be portable.

Restrictions in calling Ada from other languages

- The Ada subprogram called must be a library subprogram or a subprogram declared in the interface of a library package.
- The subprogram called must not use Ada tasking or interact with other Ada code that involves tasking.
- The Ada subprogram will follow TeleGen2 internal calling conventions. If it is called from assembly code, then the calling code must reflect those conventions.
- The subprogram must not be dependent on the prior execution of any Ada elaboration code, unless the user can guarantee that such code has been properly executed.

Working around elaboration code

Elaboration code is compiler-generated code that mainly allocates storage for dynamically sized data objects and assigns initial values to things that require them. The only reliable way to ensure that all necessary elaboration code has been executed is to be executing out of an Ada main program. In that case, the binder component of the Ada compiler will have generated a sequence of calls to the elaboration procedures of all Ada compilation units in the program. That sequence of calls will be executed by the Ada run-time system immediately prior to invocation of the main program.

If an application can be realized as an Ada main program that happens to call code written in another language, then that other code can make calls to Ada, observing other restrictions, without having to worry about elaboration code.

However, if the application requires that the main program be written in another language, then execution of Ada elaboration code is likely to be a problem.

In general, the only types of Ada subprograms that can be safely called from other languages when the application is not controlled by an Ada main program are library procedures and functions that depend only on their parameters. It is possible to handle more general cases if the application code arranges to call the required Ada elaboration procedures at startup time. However, this is not a recommended practice, and documentation of how to do it is beyond the scope of this manual. If you have a requirement for this type of usage, please contact a customer service representative.

2.1.4. Machine code insertions

The term *machine code insertions* refers to the concept of including target machine instructions directly into the high-level Ada program. The other mechanism for including target instructions into a program is to call a routine coded in assembly language. That routine is accessed by using pragma Interface (assembly).

The primary advantage of machine code insertions (MCI) over calling an assembly language procedure is the ability to directly inline the MCI instructions. MCI instructions in Ada are not as convenient or easy to use as normal assemblers, but they can be used to provide maximum speed efficiency in accessing low level operations. MCI instructions are most useful for inserting special instructions on the target which are not normally produced by an Ada program, that is, interrupt_disable, etc.

In Ada, MCI instructions must be placed in an MCI procedure with no other normal Ada instructions in that procedure (refer to Section 13.8 of the LRM). That procedure can have any class of Ada parameters. The procedure should be marked pragma Inline for the MCI instructions to be directly inserted into the calling code. However, to get an MCI procedure inlined, there must also be a pragma Interface_Information indicating an "mci" mechanism refer to Section 2.2.8.1.3). This pragma is obeyed at all levels of optimization for MCI procedures.

The user can further improve the efficiency of the inlined MCI procedure by adding more information using pragma Interface_Information. This allows the user to dictate to the compiler to pass the parameters in specific registers. Pragma Interface_Information also allows the user to indicate to the compiler which registers are "clobbered" (that is, used destructively and not restored). The compiler will properly preserve any live values in those clobbered registers across the "call." The net result of using pragmas Inline and Interface_Information on an MCI procedure is the direct insertion of the desired instructions using the proper registers into the code stream of the caller, thus achieving maximum efficiency.

If the compiler is unable to inline the MCI procedure, a warning will be issued and a normal call produced. The MCI instructions will be produced inside a normal procedure so that code will work; however, users are encouraged to correct the inline warning since the primary usage for MCI procedures is direct insertion into the normal Ada code stream.

2.1.4.1. User and compiler assumptions

TeleGen2 machine code insertions are "what you see is what you get." That is, the only instructions that are generated are those dictated by the user in code statements in the MCI procedure. The only exception is the pseudo instruction, LOAD_ADDRESS (refer to the example in Section 2.1.4.4). The other overhead

may be instructions necessary to get the actual arguments for the call into the user-dictated locations for the parameters (that is, loading into the proper register). That overhead is not significantly different from using a normal instruction in code generation.

Any use of foreign code, either through pragma Interface or MCI procedures, has the potential of making a program erroneous. The compiler can do nothing to prevent this. In the implementation of MCI instructions, it is assumed that gained flexibility and efficiency take priority over gained "safety." However, the mechanisms that provide this flexibility should be explicit, so the user can determine what is occurring.

Basic assumptions about the use of MCI procedures:

- The user needs to understand the calling conventions sufficiently so as to not violate the stack structure. User specified "clobbered" registers are always protected by the compiler.

- It is the user's responsibility to assure that the stack, registers, and system data structures are accessed correctly by the code. For example, the compiler does not check whether *in* mode parameters are modified by code statements. The compiler also does not check whether other registers are clobbered besides those specified by the user.

2.1.4.2. Ada LRM definitions for machine code insertions

Section 13.8 of the Ada Language Reference Manual defines requirements and syntax for Ada machine code insertions.

A machine code insertion is achieved in Ada by calling a procedure containing code statements. A procedure body, if it contains a code statement, may contain *only* code statements; no other Ada statement, such as *return*, is allowed.

A code statement is a fully qualified record aggregate. The base type of the aggregate must be defined in the predefined package Machine_Code. Although restrictions may be placed on the expressions allowed as components of the aggregate, it has the same syntax as that which would appear on the right hand side of a record assignment.

2.1.4.2.1. Machine code insertion procedures

The declarative part of a machine code insertion procedure may contain only a use clause; no local data may be defined.

An MCI procedure can have any number of parameters of any mode. The mode (*in*, *out*, or *in out*) is not enforced by the compiler, but must be observed by the user. If the user specifies that the parameters are passed in specific registers, then those parameters should be accessed through those registers. *Note:* All

parameters must be passed in registers. If the user does not indicate the passing parameters in registers, the parameters are passed using the same conventions as for normal Ada subprograms (refer to Section 2.1.3.2). The offset of that parameter from the frame pointer can be obtained using 'Offset. If the MCI procedure is inlined, but has not specified the parameters in registers, then the actuals are evaluated and placed on the caller stack, where again 'Offset can be used to get the displacement from the frame pointer.

If any compiler-generated parameters exist, such as a descriptor for an unconstrained array, the user must know its format and location as well.

Please note that this mechanism allows uniform reference to the parameters if the MCI procedure is inlined, and if register parameters are specified. Also note that the most efficient usage of the MCI instructions will be obtained by using Inline and register parameters.

2.1.4.2.2. Implementation-dependent attribute to access Ada objects

For MCI users who need to access Ada objects other than register parameters, two attributes are utilized, 'Address and 'Offset. These attributes allow you to access compiler information on the location of variables. 'Address is a language-defined attribute that has implementation-specific characteristics; 'Offset is an implementation-defined attribute.

'Address This attribute is normally used to access some global control variable or composite structure. 'Address is also used in conjunction with local labels. See details in the following sections on usage of 'Address in the actual code statements. Note that no special code is generated automatically; this attribute simply provides the appropriate value for the absolute address.

'Offset This attribute yields the offset of an Ada object from its parent frame. For a global object, this is the offset from the base of the compilation unit data section (although 'Address is the preferred way to access globals). For objects inside subprograms, 'Offset yields the offset in the local stack frame. This is primarily for usage with parameters that are not passed in registers. A secondary usage is to code an MCI "function" where an Ada function is wrapped around an MCI procedure declaration and then calls the MCI procedure with inlining. provides an efficient way to overcome the language limitation that MCI subprograms can only be procedures.

2.1.4.3. Code statements

2.1.4.3.1. Requirements

A code statement is an aggregate of a record type specified in package Machine_Code, which is implementation supplied and cannot be modified by the user (similar to package System). In general, no limitations are placed on the expressions appearing in a record aggregate; in particular, they do not have to be static. In the case of machine code aggregates, however, the components of an aggregate *do* have to be static. The exception is 'Address, since it is part of the compiled code prior to run time.

Code statement aggregates are treated very much like other record aggregates, with the additional restriction that the component expressions must be static and yield a literal of a discrete type. This limits each component in the aggregate to one of

- an enumeration literal,
- an integer literal,
- a constant or named number,
- an object_name'Address for an object.
- an object_name'Offset, or
- an expression consisting of the above and using only predefined operators.

This list is consistent with that given in Section 4.9 of the LRM, except that

- user-defined functions cannot appear,
- implementation-defined attributes are allowed, and
- each expression must be statically computed (not just computable).

A compile-time error message is issued if any of these conditions are violated.

2.1.4.3.2. Syntax

The LRM defines Machine_Code as the package where the code statement record type(s) are declared. In addition, the package defines enumeration types describing the target registers and addressing modes. Package Machine_Code does not have a body. The specification of this package is provided in a file in the product run-time source files directory. and it is printed in the next section.

All possible code statements are defined by a single variant record, MCI, with one discriminant, the assembly instruction. In general, a code statement is of the form

```
MCI'(<opcode>,<operand>,...,<operand>);
```

where <opcode> is an enumeral of the type Instruction_Name and <operand> is an attribute of the MCI record specification. The number of operands depends

on the instruction.

An MCI record takes as a discriminant one of a number of instruction names. The type MCI is defined near the end of package Machine_Code, which is listed in the product documentation.

2.1.5. Interrupts

The Ada LRM provides for interrupt handlers written in Ada. The approach is to associate a task entry with an interrupt source by means of an address clause. Such an entry is referred to as an *interrupt entry* (LRM 13.5.1). A task containing an interrupt entry is referred to in this section as an *interrupt task*. When an interrupt occurs, it is handled as if an entry call had been made by the hardware to the entry associated with that interrupt. For example (according to the LRM)

```
task Interrupt_Handler is
  entry Done;
  for Done use at 16#40#; -- Assume that System.Address
                          -- is an integer type
end Interrupt_Handler;
```

In this example, the interrupt entry Done is associated with the interrupt vector at hexadecimal address 40. When a physical device causes an interrupt through that vector, an entry call is made to Done, which can *handle* the interrupt in an accept statement.

The AEE provides the facilities required by the LRM and goes substantially beyond those requirements to meet the needs of realistic systems. This section describes the interrupt-related facilities of the AEE and contrasts them with the minimal mechanism defined by the LRM.

2.1.5.1. Programmer interface

Package Interrupt, in the run-time sublibrary, provides types and subprograms that allow you to define the source of an interrupt. The remainder of this section describes the types and subprograms provided by the package Interrupt.

In the TeleGen2 approach, the address clause designating an interrupt entry refers to the address of an interrupt descriptor rather than to the address of the physical interrupt source. The Interrupt package provides a private descriptor type for this purpose.

```
type Descriptor is private;
```

If a suitable descriptor object of this type is declared, the LRM example then appears as follows.

```
Device: Interrupt.Descriptor;

task Interrupt_Handler is
  entry Done;
  for Done use at Device'Address;
end Interrupt_Handler;
```

2.1.5.1.1. Software interrupts

For software interrupts, package Interrupt provides a type to represent the signal associated with the interrupt.

```
subtype Interrupt_Identification is Signal;
```

The function Source takes a parameter designating a signal, allocates an interrupt descriptor, associates that descriptor with the vector address, and returns the descriptor.

```
function Source (Vector: in Interrupt_Identification) return Descriptor
```

A call to this function can be used to initialize a Descriptor object, such as Actuator in the following example.

```
Actuator: Interrupt.Descriptor := Interrupt.Source (Interrupt.Sighup);
```

2.1.5.1.2. A simple example

In the following example, a user-implemented Interrupt_Setup package localizes the descriptor objects and their software interrupt sources.

```
with Interrupt;
package Interrupt_Setup is

  Actuator: Interrupt.Descriptor :=
            Interrupt.Source(Interrupt.Sighup);

end Interrupt_Setup;
```

A program might then contain a driver task that provides the handler (interrupt entry) for this interrupt.

```
with Interrupt_Setup;
procedure Example_Application is

  task Actuator_Driver is
    entry Device_Ready;
    for Device_Ready use at Interrupt_Setup.Actuator'Address;
  end Actuator_Driver;

  -- ...
begin
  null;
end Example_Application;
```

The entry Device_Ready handles the software interrupt when the signal is delivered to the program.

2.1.5.1.3. Interrupts as conditional entry calls

The LRM allows the entry call representing the interrupt to be in an ordinary entry call, a timed entry call, or a conditional entry call. In TeleGen2's interrupt facility, interrupts are represented by conditional entry calls. If the interrupt handler is not ready to service the interrupt when it occurs, a backup handler in a *failure task* is invoked instead. Here is how this conditional entry call to the preceding example would look if represented in Ada.

```
select
  Actuator_Driver.Device_Ready;
else
  Failure.Device_Ready_Handler;
end select;
```

Failure entries are handled as interrupt entries in that they are associated through address clauses with failure handlers. The package Interrupt provides the relevant type.

```
type Failure_Descriptor is private;
```

An additional version of the Source function allows a program to explicitly designate a failure handler when an interrupt descriptor is created.

```
function Source (Vector:  Interrupt_Identification;
                 Failure: Failure_Descriptor)
                 return Descriptor;
```

When no explicit failure handler is designated (as in the versions of the function Source introduced earlier), a default failure handler defined within the Interrupt package is assigned to the newly created interrupt descriptor. The default failure handler will call the procedure TD_Interrupt_Manager.Unhandled_Interrupt in the environment module.

This procedure is a subunit of package TD_Interrupt_Manager and can be replaced by the user. The source for the subunit can be found in the run-time library source files directory. The default implementation calls the unix "-psignal" utility to print a message indicating that a signal was delivered to the process but not handled.

The following is an example of a user-defined failure handler declaration.

```
with Interrupt;
package Failure_Example is

   Failure_Desc : Interrupt.Failure_Descriptor;
   Primary_Desc : Interrupt.Descriptor
            := Interrupt.Source( Interrupt.Sighup, Failure_Desc );

   task Failure_Handler is
```

```
      entry Unhandled_Interrupt;
      for Unhandled_Interrupt use at Failure_Desc'Address;
    end Failure_Handler;

    task Primary_Handler is
      entry Interrupt_Occurred;
      for Interrupt_Occurred use at Primary_Desc'Address;
    end Primary_Handler;

end Failure_Example;

package body Failure_Example is

    task body Failure_Handler is
    begin
      loop
        accept Unhandled_Interrupt do
 -- actions for handling interrupt when primary server is
 -- unavailable
        end accept;
      end loop;
    end Failure_Handler;

    task body Primary_Handler is
    begin
      loop
        accept Interrupt_Occurred do
 -- actions required to handle interrupt
        end Interrupt_Occurred;
      end loop;
    end Primary_Handler;

 end Failure_Example;
```

2.1.5.1.4. Changing the failure handler

The Interrupt package provides a procedure Configure that allows the association between an interrupt descriptor and a failure handler to be changed. The user can therefore have customized failure handlers, such as a failure handler for startups that ignores interrupts and a real failure handler for run-time failures. The failure handler can be changed as many times as necessary. Procedure Configure has the following interface.

```
procedure Configure (Which: Descriptor;
                     Failure: Failure_Descriptor);
```

2.1.5.1.5. Optimized interrupt entries

The facilities described so far are sufficient to implement interrupt handlers in Ada. However, the process of handling an interrupt in this fashion is potentially complicated and time-consuming. Ada does not restrict the language features that

can be used inside the body of an accept statement. Therefore, an interrupt handler could contain entry calls to other tasks or even delay statements. Furthermore, in the general case, a full Ada context switch must be made to the interrupt handler task and then a full context switch back to the interrupted task (or potentially some other ready task) when the rendezvous is completed.

In some cases, the properties of fully general Ada interrupt handlers may suit the intended application. In other cases, however, it may be necessary to trade a reduction in generality for an increase in performance in order to meet application requirements. The AEE addresses these needs by allowing programmers to select one of two optimized constructs by which task entries can handle interrupts.

synchronization optimizations — The interrupt serves only to cause the handler task to become *ready to execute* without requiring an actual context switch as part of servicing the interrupt.

function-mapped optimizations — All processing associated with handling the interrupt occurs during the rendezvous (in the body of the accept statement) and no interactions with other tasks occur during the rendezvous.

2.1.5.1.6. Synchronization optimizations

A synchronization optimization corresponds to having an empty accept body that simply puts the interrupt handler on a ready queue. This optimization is always applied when appropriate, without explicit programmer request. In the following example, occurrence of the sighup signal causes Actuator_Driver to be placed in the ready queue.

```
task body Actuator_Driver is
begin
  accept Device_Ready;
  -- Actions responding to the device-ready signal
end Actuator_Driver;
```

Actuator_Driver is activated when its priority is higher than the priorities of competing tasks.

2.1.5.1.7. Function-mapped optimizations

In a function-mapped optimization, all the interrupt handling work is done inside the accept body during the rendezvous. When this optimization is invoked, the compiler maps the accept body into a function that can be directly called from the signal handler. This kind of optimization is restricted to accept statements that do not interact with other tasks during the rendezvous. The following fragment illustrates the use of pragma Interrupt.

```
task body Actuator_Driver is
begin
  pragma Interrupt (Function_Mapping);
  accept Device_Ready do
    -- Actions responding to the Device_Ready interrupt.
  end Device_Ready;
end Actuator_Driver;
```

The pragma Interrupt applies to the statement immediately following it, which must be one of the following three constructs:

1. A simple accept statement, as described in the preceding.

2. A while loop directly enclosing only a single accept statement, discussed in the following.

3. A select statement that includes an interrupt accept alternative.

For reasons related to the loop optimization discussed in the following, the server task with a function-mapped accept cannot have a user-specified priority.

The body of the accept statement handling the interrupt is executed in the environment of the interrupted current task. Note that the function-mapped body acts much like a classic interrupt procedure and requires no context switch even though it acts in the proper lexical environment.

The interrupt server often executes a small or null amount of non-handler code between accepting interrupt entry calls. The interrupt support is designed to take advantage of this occurrence to minimize latency in the driver and execute another handler with the minimum number of task switches. The best special case for this is an accept statement directly embedded inside a loop. For instance, the actuator driver is presented with a buffer of actuator commands. The driver contains a loop that waits on successive occurrences of the Device_Ready interrupt and issues commands out of the buffer. The function-mapping optimization caters to this possibility as well.

The following fragment shows the second class of constructs to which the function-mapping optimization can be applied—a while loop that contains only an accept statement for an interrupt entry. The accept statement must meet the constraint described earlier (i.e., contain no interactions with other tasks).

```
task body Actuator_Driver is
begin
  -- ...
  pragma Interrupt (Function_mapping);
  while More_Commands loop
    accept Device_Ready do
      -- Issue the next command
```

```
    end Device_Ready;
  end loop;
  -- ...
end Actuator_Driver;
```

2.1.5.1.8. Rationale for function-mapped optimizations

This section explains why the accept body is mapped into a function rather than into a procedure. The mapping is done so that the value of the loop control expression can be returned as the value of the function. The function value therefore indicates whether there will be another cycle through the loop to wait again for a Device_Ready interrupt. If the function value is True, the loop continues; if the value is False, the loop terminates.

This information about loop continuation can be used productively in the interrupt handling process. If the loop is to continue, it is highly desirable to return as quickly as possible to the interrupt accept statement so as to be ready for the next interrupt. Therefore, the rescheduling that ordinarily occurs on exit from an accept body is bypassed in this case. That is, the program continues executing at the priority of the interrupt handler until it reaches the accept statement again. Recall that the Ada LRM specifies that interrupt handler priorities are always more urgent than any other priorities expressible in Ada.

On the other hand, if the loop is to complete after the execution of the accept body, then the rescheduling on exit from the accept body is necessary and is done. These effects of the function value on the interrupt handling process are explained further in the next section.

A function-mapped optimization for a simple accept statement, that is, one not in a while loop, produces a function that always returns False. Such an optimization can dramatically reduce the time required to begin executing the interrupt handler. After the function-mapped routine has handled the immediate interrupt processing, it causes a rescheduling event to activate the server task. This preemption is justified by the assumption that only small amounts of non-interrupt code will need to be executed before another handler will be reached. This approach expedites reaching the next occurrence of the handler, which reduces the period in which interrupts from the device must be delivered to the failure handler.

This subtle treatment of priorities in order to improve interrupt throughput is only legal if no explicit pragma Priority applies to the interrupt handler task. In that case, the LRM allows the implementation to determine the priority of that task, and this optimization is legal. If an explicit priority does apply to an interrupt handling task, this optimization pragma is ignored by the compiler.

2.1.5.2. Rationale behind the TeleGen2 interrupt strategy

This section discusses several of the important attributes of the TeleGen2 approach to interrupt handling and the benefits that result from that approach. The attributes discussed are the following

- Use of interrupt descriptors in address clauses instead of physical vector addresses.
- Treatment of interrupts as conditional entry calls.
- Provision for optimized interrupt entry calls.

2.1.5.2.1. Interrupt descriptors

This fundamental aspect of the TeleGen2 approach, in addition to having its own merits, is crucial to supporting the other attributes without making Ada language extensions. The key result of the descriptor approach is to allow separation between the declaration of the entry (and the associated address clause) from the specification of the corresponding interrupt source. This separation:

- Promotes portability and maintainability of interrupt software because the physical interrupt source can be *hidden* from the driver (interrupt handler task), which usually does not need to depend on that information.
- Eases the construction of test frames for interrupt software in either host environments or special developmental target configurations. During testing, different vector addresses or software interrupts may be substituted for hardware interrupts. Like the LRM approach, the TeleGen2 approach preserves the option of calling interrupt entries from ordinary Ada tasks, which may be simulating the operation of hardware.
- Allows delayed associations between handler tasks and interrupt sources, such as to ensure that hardware can be put in a suitable state before the association occurs.

2.1.5.2.2. Interrupts as conditional entry calls

The obvious benefit of this approach is greater robustness in the face of hardware failures (such as unanticipated interrupts) or software failures (such as drivers that are not engineered properly for interrupt rates that occur in practice). Default failure handlers are automatically provided if programmers choose not to specify failure handlers explicitly. Also, failure handlers for a given interrupt can be dynamically changed as the situation warrants.

2.2. Annotations

TeleGen2 compiles the full ANSI Ada language as defined by the *Reference Manual for the Ada Programming Language* (LRM) (ANSI/MIL-STD-1815A). The following sections describe the portions of the language that are designated by the LRM as implementation dependent for the compiler and run-time environment.

The information is presented in the order in which it appears in the LRM. In general, however, only those language features that are not fully implemented by the current release of TeleGen2 or that require clarification are included. The features that are optional or that are implementation dependent, on the other hand, are described in detail. Particularly relevant are the sections annotating LRM Chapter 13 (Representation Clauses and Implementation-Dependent Features) and Appendix F (Implementation-Dependent Characteristics).

2.2.1. LRM Chapter 2 - Lexical Elements

[LRM 2.1] Character Set

The host and target character set is the ASCII character set.

[LRM 2.2] Lexical Elements, Separators, and Delimiters

The maximum number of characters on an Ada source line is 200.

[LRM 2.8] Pragmas

TeleGen2 implements all language-defined pragmas *except* pragma Optimize. If pragma Optimize is included in Ada source, the pragma will have no effect. Optimization is implemented by using pragma Inline and the optimizer.

Limited support is available for pragmas Memory_Size, Storage_Unit, and System_Name; that is, these pragmas are allowed if the argument is the same as the value specified in the System package.

Pragmas Page and List are supported in the context of source/error listings, as described in the Compiler/Linker chapter of the *TeleGen2 User Guide*.

2.2.2. LRM Chapter 3 - Declarations and Types

[LRM 3.2.1] Object Declarations

TeleGen2 does not produce warning messages about the use of uninitialized variables. The compiler will not reject a program merely because it contains such variables.

[LRM 3.5.1] Enumeration Types

The maximum number of elements in an enumeration type is 2147483647 (the value of System.Max_Int). This maximum can be realized only if generation of the image table for the type has been deferred, and there are no references in the program that would cause the image table to be generated. Deferral of image table generation for an enumeration type, P, is requested by the statement

```
pragma Images (P, Deferred);
```

Section 2.2.8.1, "Implementation-defined pragmas," describes more about pragma Images.

[LRM 3.5.4] Integer Types

There are three predefined integer types: Short_Short_Integer, Short_Integer, and Integer. The attributes of these types are shown in Table 6. Note that using explicit integer type definitions instead of predefined integer types should result in more portable code.

Table 6. Attributes of predefined integer types

Attribute	Type		
	Short_Short_Integer	Short_Integer	Integer
'First	-128	-32768	-2147483648
'Last	127	32767	2147483647
'Size	8	16	32
'Width	4	6	11

[LRM 3.5.8] Operations of Floating Point Types

There are two predefined floating point types: Float and Long_Float. The attributes of types Float and Long_Float are shown in Table 7. This floating point facility is based on the IEEE standard for 32-bit and 64-bit numbers. Note that using explicit real type definitions should lead to more portable code.

The type Short_Float is not implemented.

Table 7. Attributes of predefined floating point types

Attribute	Type	
	Float	Long_Float
'Machine_Overflows	TRUE	TRUE
'Machine_Rounds	TRUE	TRUE
'Machine_Radix	2	2
'Machine_Mantissa	24	53
'Machine_Emax	128	1024
'Machine_Emin	-125	-1021
'Mantissa	21	51
'Digits	6	15
'Size	32	64
'Emax	84	204
'Safe_Emax	125	1021
'Epsilon	9.53674E-07	8.88178419700125E-16
'Safe_Large	4.25353E+37	2.24711641857789E+307
'Safe_Small	1.17549E-38	2.22507385850720E-308
'Large	1.93428E+25	2.57110087081438E+61
'Small	2.58494E-26	1.94469227433161E-62

2.2.3. LRM Chapter 4 - Names and Expressions

[LRM 4.10] Universal Expressions

There is no limit on the accuracy of real literal expressions. Real literal expressions are computed using an arbitrary-precision arithmetic package.

2.2.4. LRM Chapter 9 - Tasks

[LRM 9.6] Delay Statements, Duration, and Time

This implementation uses 32-bit fixed point numbers to represent the type Duration. The attributes of the type Duration are shown in Table 8.

Table 8. Attributes of type Duration

Attribute	Value
'Delta	2**(-14)
'First	-86400.0
'Last	86400.0

[LRM 9.8] Priorities

Sixty-four levels of priority are available to associate with tasks through pragma Priority. The predefined subtype Priority is specified in the package System as

```
subtype Priority is Integer range 0..63;
```

Currently the priority assigned to tasks without a pragma Priority specification is 31; that is,

```
(System.Priority'First + System.Priority'Last)  / 2
```

[LRM 9.11] Shared Variables

The restrictions on shared variables are only those specified in the LRM.

2.2.5. LRM Chapter 10 - Program Structure and Compilation Issues

[LRM 10.1] Compilation Units - Library Units

All main programs are assumed to be parameterless procedures or functions that return an integer result type.

2.2.6. LRM Chapter 11 - Exceptions

[LRM 11.1] Exception Declarations

Numeric_Error is raised for integer or floating point overflow and for divide-by-zero situations. Floating point underflow yields a result of zero without raising an exception.

Program_Error and Storage_Error are raised by those situations specified in LRM Section 11.1.

2.2.7. LRM Chapter 13 - Implementation-Dependent Features

The current release of TeleGen2 supports most LRM Chapter 13 facilities, as summarized below. The sections that follow the summary describe the LRM Chapter 13 facilities that are either not implemented or that require explanation. Facilities implemented exactly as described in the LRM are not mentioned.

13.1 Representation clauses	Supported, except as indicated in the following (LRM 13.2-13.5). Pragma Pack is supported except for dynamically sized components.
13.2 Length clauses	Supported: 'Size 'Storage_Size for collections 'Storage_Size for task activation 'Small for fixed-point types
13.3 Enumeration representation clauses	Supported except for type Boolean or types derived from Boolean. Users can easily define a non-Boolean enumeration type and assign a representation clause to it.
13.4 Record representation clauses	Supported except for records with dynamically sized components.
13.5 Address clauses	Supported for: objects (including task objects). Not supported for: packages, subprograms, or task units.
13.5.1 Interrupts	For interrupt entries, the address of a TeleGen2-defined interrupt descriptor can be given.
13.7 Package System	Conforms closely to the LRM model. The specification is listed in Section 2.2.7.7.
13.7.1 System-dependent named numbers	Defined in the specification of package System (Section 2.2.7.7).
13.7.2 Representation attributes	Implemented as described in the LRM except that: 'Address for packages is unsupported. 'Address of a constant yields a null address.
13.7.3 Representation attributes of real types	Shown in Table 7 in Section 2.2.2.
13.8 Machine code insertions	Fully supported. The TeleGen2 implementation defines an attribute, 'Offset, that, along with the language-defined attribute 'Address, allows addresses of objects and offsets of data items to be specified in stack frames. Section 2.1.4, "Machine code insertions," describes the implementation

	and use of machine code insertions.
13.9 Interface to other languages	Pragma Interface is supported for assembly, FORTRAN, Pascal, and C. Section 2.1.3, "Pragma Interface," describes the implementation and use of pragma Interface.
13.10 Unchecked programming	Supported except as noted in the following (LRM 13.10.2).
13.10.2 Unchecked type conversions	Supported except for the case where the destination type is an unconstrained record or array.

2.2.7.1. The TeleGen2 Implementation of pragma Pack

Pragma Pack with Boolean arrays

You can pack Boolean arrays by the use of pragma Pack. The compiler allocates 8 bits for a single Boolean, 8 bits for a component of an unpacked Boolean array, and 1 bit for a component of a packed Boolean array. The first figure illustrates the layout of an unpacked Boolean array; the figure that follows it illustrates a packed Boolean array.

Unpacked Boolean array

```
Unpacked_Bool_Arr_Type is array (Natural range 0..1) of Boolean
U_B_Arr: Unpacked_Bool_Arr_Type := (True,False);
```

Packed Boolean array

```
Packed_Bool_Arr_Type is array (Natural range 0..6) of Boolean;
    pragma Pack (Packed_Bool_Arr_Type);
    P_B_Arr: Packed_Bool_Arr_Type :=
      (P_B_Arr(0) => True, P_B_Arr(5) => True, others => False);
```

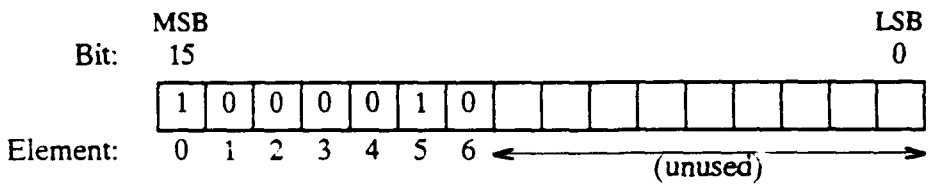


Figure 1. Packed and unpacked Boolean arrays

2.2.7.2. [LRM 13.2] Length Clauses

A length clause specifies an amount of storage associated with a type. The sections that follow describe how length clauses are supported in this implementation of TeleGen2 and how to use length clauses effectively within the context of TeleGen2.

2.2.7.2.1. (a) Specifying size: T'Size

The prefix T denotes an object. The size specification must allow for enough storage space to accommodate every allowable value of these objects. The constraints on the object and on its subcomponents (if any) must be static. For an unconstrained array type, the index subtypes must also be static.

For this implementation, Min_Size is the smallest number of bits logically required to hold any value in the range; no sign bit is allocated for non-negative ranges. Biased representations are not supported; e.g., a range of 100 .. 101 requires 7 bits, not 1.

Caution: In the current release, using a size clause for a discrete type may cause inefficient code to be generated. For example, given

```
type Nibble is range 0 .. 15;
for Nibble'Size use 4;
```

each object of type Nibble will occupy only 4 bits, and relatively expensive bit-field instructions will be used for operations on Nibbles. (A single declared object of type Nibble will be aligned on a storage-unit boundary, however.)

For floating-point and access types, a size clause has no effect on the representation.

For composite (array or record) types, a size clause acts like an implicit pragma Pack, followed by a check that the resulting size is no greater than the requested size. Note that the composite type will be packed whether or not it is necessary to meet the requested size. The size clause for a record must be a multiple of storage units.

2.2.7.2.2. (b) Specifying collection size: T'Storage_Size

A collection is the entire set of objects created by evaluation of allocators for an access type.

The prefix T denotes an access type. Given an access type Acc_Type, a length clause for a collection allocated using Acc_Type objects might look like the following.

```
for Acc_Type'Storage_Size use 64;
```

In TeleGen2, the above length clause allocates from the heap 64 bytes of contiguous memory for objects created by Acc_Type allocators. Every time a new object is created, it is put into the remaining free part of the memory allocated for the collection, provided there is adequate space remaining in the collection. Otherwise, a Storage_Error is raised.

Keeping the objects in a contiguous span of memory allows system storage reclamation routines to deallocate and manage the space when it is no longer needed. Pragma Controlled can prevent the deallocation of a specified collection of objects. Objects can be explicitly deallocated by calling the Unchecked_Deallocation procedure instantiated for the object and access types.

Given an access type which does not have a length clause specified, the 'Storage_Size attribute will return a value of 0.

Header record

In this configuration of TeleGen2, information needed to manage storage blocks in a collection is stored in a collection header adjacent to the collection, in addition to the value specified in the length clause.

Minimum Size

When an object is deallocated from a collection, a record containing link and size information for the space is put in the deallocated space as a placeholder. This enables the space to be located and reallocated. The space allocated for an object must therefore have the minimum size needed for the placeholder record. For this TeleGen2 configuration, this minimum size is the sum of the sizes of an access type and an integer type.

Dynamically Sized Objects

When a dynamically-sized object is allocated, a record accompanies it to keep track of the size of the object for when it is put on the free list. The record is used to set the size field in the placeholder record since compaction may modify the value.

Size Expressions

Instead of specifying an integer in the length clause, you can use an expression to specify storage for a given number of objects. For example, suppose an access type Dict_Ref references a record Symbol_Rec containing five fields:

```
type Tag is String(1..8);

type Symbol_Rec;
type Dict_Ref is access Symbol_Rec;

type Symbol_Rec is
  record
    Left   : Dict_Ref;
    Right  : Dict_Ref;
    Parent : Dict_Ref;
    Value  : Integer;
    Key    : Tag;
  end record;
```

To allocate 10 Symbol_Rec objects, you could use an expression such as

```
for Dict_Ref'Storage_Size use ((Symbol_Rec'Size * 10)+X);
```

where X is the extra space needed for the header record. (Symbol_Rec is obviously larger than the minimum size required, which is equivalent to one access type and one integer.)

In another implementation, Symbol_Rec might be a variant record that uses a variable length for the string Key.

```
type Symbol_Rec(Last : Natural :=0) is
  record
    Left   : Dict_Ref;
    Right  : Dict_Ref;
    Parent : Dict_Ref;
    Value  : Integer;
    Key    : String(1..Last);
  end record;
```

In this case, Symbol_Rec objects would be dynamically sized depending on the length of the string for Key. Using a length clause for Dict_Ref as above would then be illegal since Symbol_Rec'Size cannot be consistently determined. A length clause for Symbol_Rec objects, as described in (a) above, would be illegal since not all components of Symbol_Rec are static. As defined, a Symbol_Rec object could conceivably have a Key string with Integer'Last number of characters.

2.2.7.2.3. (c) Specifying storage for task activation: T'Storage_Size

The prefix T denotes a task type. A length clause for a task type specifies the number of storage units to be reserved for an activation of a task of the type.

2.2.7.2.4. (d) Specifying 'Small for fixed point types: T'Small

Small is the absolute precision (a positive real number) while the prefix T denotes the first named subtype of a fixed point type. Elaboration of a real type defines a set of model numbers. T'Small is generally a power of 2, and model numbers are generally multiples of this number so that they can be represented exactly on a binary machine. All other real values are defined in terms of model numbers having explicit error bounds. For example, consider a type Fixed.

```
type Fixed is delta 0.25 range -10.0 .. 10.0;

Fixed'Small = 0.25          -- A power of 2

3.0 = 12 * 0.25             -- A model number but not a power of 2
```

The value of the expression of the length clause must not be greater than the delta of the first named subtype. The effect of the length clause is to use this value of 'Small for the representation of values of the fixed point base type. The length clause thereby also affects the amount of storage for objects that have this type.

If a length clause is not used, for model numbers defined by a fixed point constraint, the value of Small is defined as the largest power of two that is not greater than the delta of the fixed accuracy definition.

If a length clause is used, the model numbers are multiples of the specified value for Small. For this configuration of TeleGen2, the specified value must be (mathematically) equal to either an exact integer or the reciprocal of an exact integer.

Examples of model numbers:

```
1.0, 2.0, 3.0, 4.0, . . .       -- legal
0.5, 1.0/3.0, 0.25, 1.0/3600.0 -- legal
2.5, 2.0/3.0, 0.3               -- illegal
```

2.2.7.3. [LRM 13.3] Enumeration Representation Clauses

Enumeration representation clauses are supported, except for Boolean types.

Note: Be aware that use of such clauses may introduce considerable overhead into many operations that involve the associated type. Such operations include indexing an array by an element of the type, or computing the 'Pos, 'Pred, or 'Succ attributes for values of the type.

2.2.7.4. [LRM 13.4] Record Representation Clauses

Since record components are subject to rearrangement by the compiler, you must use representation clauses to guarantee a particular layout. Such clauses are subject to the following constraints:

- Each component of the record must be specified with a component clause.
- The alignment of the record is restricted to mods 1, 2, and 4; byte, word, and long-word aligned, respectively.
- Bits are ordered with bit zero as the most significant bit.
- Floating point and fixed point components may not cross word boundaries.

Here is a simple example showing how the layout of a record can be specified by using representation clauses.

```
package Repspec_Example is
  Bits : constant := 1;
  Word : constant := 4;
```

```
        type Five is range 0 .. 16#1F#;
        type Seventeen is range 0 .. 16#1FFFF#;
        type Nine is range 0 .. 511;

        type Record_Layout_Type is record
            Element1 : Seventeen;
            Element2 : Five;
            Element3 : Boolean;
            Element4 : Nine;
          end record;

        for Record_Layout_Type use record at mod 2;
            Element1 at 0*Word range 0 .. 16;
            Element2 at 0*Word range 17 .. 21;
            Element3 at 0*Word range 22 .. 22;
            Element4 at 0*Word range 23 .. 31;
          end record;

        Record_Layout : Record_Layout_Type;
      end Repspec_Example;
```

2.2.7.5. [LRM 13.5] Address Clauses

The Ada compiler supports address clauses for objects and entries. Address clauses for packages and task units are not supported.

Address clauses for objects can be used to access hardware memory registers or other known memory locations. The use of address clauses is affected by the fact that the System.Address type is private. For the SUN-4 target, literal addresses are represented as integers, so an unchecked conversion must be applied to these literals before they can be passed as parameters of type System.Address. The examples in this document often assume the following declaration:

```
function Addr is new Unchecked_Conversion (Integer,System.Address);
```

This function is invoked when an address literal needs to be converted to an Address type. Naturally, user programs can implement a different convention. The following is a sample program that uses address clauses and this convention. Package System must be explicitly *with*ed when using address clauses.

```
with System;
with Unchecked_Conversion;
procedure Hardware_Access is
  function Addr is new Unchecked_Conversion (Integer, System.Address);
  Hardware_Register : integer;
  for Hardware_Register use at Addr (16#FF0000#);
begin
  ...
end Hardware_Access;
```

When using an address clause for an object with an initial value, the address clause should immediately follow the object declaration.

```
Obj: Some_Type := <init_expr>;
for Obj use at <addr_expr>;
```

This sequence allows the compiler to perform an optimization wherein it generates code to evaluate the <addr_expr> as part of the elaboration of the declaration of the object. The expression <init_expr> will then be evaluated and assigned directly to the object, which is stored at <addr_expr>. If another declaration had intervened between the object declaration and the address clause, the compiler would have had to create a temporary object to hold the initialization value before copying it into the object when the address clause is elaborated. If the object were a large composite type, the need to use a temporary could result in considerable overhead in both time and space. To optimize your applications, therefore, you are encouraged to place address clauses immediately after the relevant object declaration.

As mentioned above, arrays containing components that can be allocated in a signed or unsigned byte (8 bits) are packed, one component per byte. The following example indicates how these facts allow access to hardware byte registers:

```
with System;
with Unchecked_Conversion;
procedure Main is
  function Addr is new Unchecked_Conversion(Integer, System.Address);
  type Byte is range -128..127;
  HW_Regs : array (0..1) of Byte;
  for HW_Regs use at Addr (16#FFF310#);

  Status_Byte : constant integer := 0;
  Next_Block_Request: constant integer := 1;
  Request_Byte : Byte := 119;
  Status : Byte;

begin
  Status := HW_Regs(Status_Byte);
  HW_Regs(Next_Block_Request) := Request_Byte;
end Main;
```

Two byte hardware registers are referenced in the example above. The status byte is at location 16#FFF310# and the next block request byte is at location 16#FFF311#.

Function Addr takes an integer as its argument. Integer'Last is 16#7FFFFFFF#, but there are certainly addresses greater than Integer'Last. Those addresses with the high bit set, such as FFFA0000, cannot be represented as a positive integer. Thus, for addresses with the high bit set, the address should be computed as the negation of the 2's complement of the desired address. According to this method, the correct representation of the sample address above would be Addr(-16#00060000#).

2.2.7.6. [LRM 13.6] Change of Representation

TeleGen2 supports changes of representation.

2.2.7.7. [LRM 13.7] The Package System

The specification of TeleGen2's implementation of package System is presented in the LRM Appendix F section at the end of this chapter, in Section 2.2.8.3.

2.2.7.8. [LRM 13.7.2] Representation Attributes

The compiler does not support 'Address for packages.

2.2.7.9. [LRM 13.7.3] Representation Attributes of Real Types

The representation attributes for the predefined floating point types are presented in Table 7.

2.2.7.10. [LRM 13.8] Machine Code Insertions

Machine code insertions, an optional feature of the Ada language, are fully supported in TeleGen2. Refer to Section 2.1.4, "Machine code insertions," for information regarding their implementation and for examples of their use.

2.2.7.11. [LRM 13.9] Interface to Other Languages

In TeleGen2, pragma Interface is supported for Assembly, C, FORTRAN, and Pascal. Refer to Section 2.1.3, "Pragma Interface," for information on the use of pragma Interface.

2.2.7.12. [LRM 13.10] Unchecked Programming

Unchecked_Conversion is allowed except when the target data subtype is an unconstrained array or record type. If the size of the source and target are static and equal, the compiler will perform an exact copy of data from the source object to the target object.

Where the sizes of source and target differ, the following rules apply.

- If the size of the source is greater than the size of the target, the high address bits will be truncated in the conversion.
- If the size of the source is less than the size of the target, the source will be moved into the low address bits of the target.

The compiler will issue a warning when Unchecked_Conversion is instantiated with unequal sizes for source and target subtype. Unchecked_Conversion between objects of different or non-static sizes will usually produce less efficient

code and should be avoided, if possible.

2.2.8. LRM Appendix F for TeleGen2

The Ada language definition allows for certain target dependencies. These dependencies must be described in the reference manual for each implementation, in an "Appendix F" that addresses each point listed in LRM Appendix F. The summary in this section constitutes Appendix F for this implementation. Points that require further clarification are addressed in sections referenced in the summary.

Implementation-dependent pragmas

Implementation-defined pragmas:

```
pragma Comment                    pragma Linkname
pragma Images                     pragma No_Suppress
pragma Interface_Information      pragma Preserve_Layc
pragma Interrupt                  pragma Suppress_All
```

Predefined pragmas with implementation-dependent characteristics:

Fully supported predefined pragmas:

```
pragma Controlled   pragma Priority
pragma Elaborate    pragma Shared
pragma Inline       pragma Suppress
```

Partly supported predefined pragmas:

```
pragma Memory_Size
pragma Storage_Unit
pragma System_Name
```

Unsupported predefined pragmas:

```
pragma Optimize
```

Implementation-dependent attributes

```
'Extended_Image   'Extended_Fore
'Extended_Value   'Subprogram_Value
'Extended_Width   'Address
'Extended_Aft     'Offset  (in MCI)
'Extended_Digits
```

'Address is not supported for packages.

Package System

Defined in Section 2.2.7.7.

Restrictions on representation clauses

Summariz .d in Section 2.2.7.

Implementation-generated names

None.

Address clause expression interpretation

An expression that appears in an object address clause is interpreted as the address of the first storage unit of the object.

Restrictions on unchecked conversions

Summarized in Section 2.2.7.

Implementation-dependent characteristics of the I/O packages

1. In Text_IO, type Count is defined as follows:

```
type Count is range 0..(2 ** 31)-2;
```

2. In Text_IO, type Field is defined as follows:

```
subtype Field is integer range 0..1000;
```

3. In Text_IO, the Form parameter of procedures Create and Open is not supported. (If you supply a Form parameter with either procedure, it is ignored.)

4. The standard library contains preinstantiated versions of Text_IO.Integer_IO for types Short_Integer and Integer, and of Text_IO.Float_IO for types Float and Long_Float. We suggest that you use the following to eliminate multiple instantiations of these packages:

```
Integer_Text_IO      Short_Integer_Text_IO
Float_Text_IO        Long_Float_Text_IO
```

2.2.8.1. Implementation-defined pragmas

There are eight implementation-defined pragmas in TeleGen2: pragmas Comment, Images, Interface_Information, Interrupt, Linkname, No_Suppress, Preserve_Layout, and Suppress_All.

2.2.8.1.1. Pragma Comment

Pragma Comment is used for embedding a comment into the object code. The syntax is:

```
pragma Comment ( <string_literal> );
```

where <string_literal> represents the characters to be embedded in the object code. Pragma Comment is allowed only within a declarative part or immediately within a package specification. Any number of comments may be entered into the object code by use of pragma Comment.

2.2.8.1.2. Pragma Images

Pragma Images controls the creation and allocation of the image and index tables for a specified enumeration type. The image table is a literal string consisting of enumeration literals catenated together. The index table is an array of integers specifying the location of each literal within the image table. The length of the index table is therefore the sum of the lengths of the literals of the enumeration type; the length of the index table is one greater than the number of literals.

The syntax of the pragma is:

```
pragma Images(<enumeration_type>, Deferred);

pragma Images(<enumeration_type>, Immediate);
```

The "Deferred" option saves space in the literal pool by not creating image and index tables for an enumeration type unless the 'Image, 'Value, or 'Width attribute for the type is used. If one of these attributes is used, the tables are generated in the literal pool of the compilation unit in which the attribute appears. If the attributes are used in more than one compilation unit, more than one set of tables is generated, eliminating the benefits of deferring the table. In this case, the "Immediate" option saves space by causing a single image table to be generated in the literal pool of the unit declaring the enumeration type. For the SUN-4, "immediate" is the default option.

For a very large enumeration type, the length of the image table will exceed Integer'Last (the maximum length of a string). In this case, using either

```
pragma Images(<enumeration_type>, Immediate);
```

or the 'Image, 'Value, or 'Width attribute for the type will result in an error message from the compiler. Therefore, use the "Deferred" option, and avoid using 'Image, 'Value, or 'Width in this case.

2.2.8.1.3. Pragma Interface_Information

The existing Ada interface pragma only allows specification of a language name. In some cases, the optimizing code generator will need more information than can be derived from the language name. Therefore there is a need for an implementation-specific pragma, Interface_Information.

There is an extended usage of this pragma for machine code insertion procedures that do not use a preceding pragma Interface. Other than that case, a pragma Interface_Information is always associated with a pragma Interface. The pragma has the following syntax:

```
pragma Interface_Information (Name,
                              Link_Name,
                              Mechanism,
                              Parameters,
                              Clobbered_Regs);
```

The parameters to the pragma are defined as follows:

```
Name            :== ada_subprogram_identifier, required
Link_Name       :== string, default = ""
Mechanism       :== string, default = ""
Parameters      :== string, default = ""
Clobbered_Regs  :== string, default = ""
```

Scope of usage

Pragma Interface_Information is allowed wherever the standard pragma Interface is allowed, and must be immediately preceded by a pragma Interface referring to the same Ada subprogram, in the same declarative part or package specification; no intervening declaration is allowed between the Interface and Interface_Information pragmas. Unlike pragma Interface, this pragma is not allowed for overloaded subprograms (it specifies information that pertain to one specific body of non-Ada code). If the user wishes to use overloaded Ada names, the Interface_Information pragma may be applied to unique renaming declarations.

The pragma is also allowed for a library unit; in that case, the pragma must occur immediately after the corresponding Interface pragma, and before any subsequent compilation unit.

This pragma may be applied to any interfaced subprogram, regardless of the language or system named in the interface pragma. The code generator is responsible for rejecting or ignoring illegal or redundant interface information. The optimizing code generator will process and check the legality of such interfaced subprograms at the time of the spec compilation, instead of waiting for an actual use of the interfaced subprogram. This will save the user from extensive recompilation of the offensive specification and all its dependents should an illegal pragma have been used.

This pragma is also used for Machine Code Insertion (MCI) procedures. In that case, the "mechanism" should be set to "mci." This allows the user to specify detailed parameter characteristics for the call and inlined call to the MCI procedure. When used in conjunction with pragma Inline, this allows the user to directly insert a minimal set of instructions into the call location.

Parameters

Name

This parameter is an Ada subprogram identifier. The rule detailed in LRM

13.9 for a subprogram named in a pragma Interface applies here as well. As explained above, the subprogram must have been named in an immediately preceding Interface pragma.

This is the only required parameter. Since the other parameters are optional, positional association can only be used if all parameters are specified, or only the rightmost ones are defaulted.

`Link_Name`

This parameter is a string literal. When specified, this parameter indicates the name the code generator must use to reference the named subprogram. This string name may contain any characters allowed in an Ada string and must be passed unchanged (in particular, not case-mapped) to the code generator. The code generator will reject names that are illegal in the particular language or system being targeted.

If this parameter is not specified, it defaults to a null string. The code generator will interpret a default link_name differently, depending on the target language/system (the default is generally the Ada name, or is derived from it, for example, "_Ada_name" for 'C' calls).

`Mechanism`

This parameter is a string literal. The only mechanism currently implemented is the "mci" mechanism used strictly in conjunction with Machine Code Insertion procedures.

`Parameters`

This parameter is a string literal. When present, this string tells the code generator where to pass each parameter. This string is interpreted as a positional aggregate where each position refers to a parameter of the interfaced subprogram. Each position may be one of the following: null, the name of a register, or the word "stack." Null arguments imply standard conventions. Thus the string "r3, stack, r5" specifies that the first parameter is to be passed in register r3, the second parameter is to be put on the stack in the parameter block (in the proper position of the second parameter), and the third parameter is to be passed in register r5.

`Clobbered_Regs`

This parameter is a string literal indicating the registers (in a comma-separated list) that are destroyed by this operation. The code generator will save anything valuable in these registers at the point of the call.

The following is a simple example of the use of pragma Interface_Information.

```
procedure Do_Something (Addr: System.Address; Len: Integer);
pragma Interface (Assembly, Do_Something);
pragma Interface_Information ( Name       => Do_Something,
                               Link_Name  => "DOIT",
                               Parameters => "00,02");
```

2.2.8.1.4. Pragma Interrupt

Pragma Interrupt is described in Section 2.1.5, "Interrupts." Please refer to Section 2.1.5.1.5, "Optimized interrupt entries," and Section 2.1.5.1.7, "Function-mapped optimizations," for a detailed discussion.

2.2.8.1.5. Pragma Linkname

Pragma Linkname is used to provide interface to any routine whose name cannot be specified by an Ada string literal. This allows access to routines whose identifiers do not conform to Ada identifier rules.

Pragma Linkname takes two arguments. The first is a subprogram name that has been previously specified in a pragma Interface statement. The second is a string literal specifying the exact link name to be employed by the code generator in emitting calls to the associated subprogram. The syntax is

```
pragma Interface ( assembly, <subprogram_name> );
pragma Linkname ( <subprogram_name>, <string_literal> );
```

If pragma Linkname does not immediately follow the pragma Interface for the associated program, a warning will be issued saying that the pragma has no effect.

A simple example of the use of pragma Linkname is

```
procedure Dummy_Access( Dummy_Arg : System.Address );
pragma Interface (assembly, Dummy_Access );
pragma Linkname (Dummy_Access, "_access");
```

Note: It is preferable to use pragma Interface_Information for this functionality. Linkname is only provided for compatibility.

2.2.8.1.6. Pragma No_Suppress

No_Suppress is a TeleGen2-defined pragma that prevents the suppression of checks within a particular scope. It can be used to override pragma Suppress in an enclosing scope. No_Suppress is particularly useful when you have a section of code that relies upon predefined checks to execute correctly, but you need to suppress checks in the rest of the compilation unit for performance reasons.

Pragma No_Suppress has the same syntax as pragma Suppress and may occur in the same places in the source. The syntax is

```
pragma  No_Suppress (<identifier> [, [ON =>] <name>]);
```

where <identifier> is the type of check you want to suppress. Checks that may be suppressed are Access_Check, Discriminant_Check, Index_Check, Length_Check, Range_Check, Division_Check, Overflow_Check, Elaboration_Check, and Storage_Check (refer to LRM 11.7).

<name> is the name of the object, type/subtype, task unit, generic unit or subprogram within which the check is to be suppressed; <name> is optional.

If neither Suppress nor No_Suppress is present in a program, checks will not be suppressed. You may override this default at the command level, by compiling the file with the -i(nhibit option and specifying with that option the type of checks you want to suppress.

If either Suppress or No_Suppress are present, the compiler uses the pragma that applies to the specific check in order to determine whether that check is to be made. If both Suppress and No_Suppress are present in the same scope, the pragma declared last takes precedence. The presence of pragma Suppress or No_Suppress in the source takes precedence over an -i(nhibit option provided during compilation.

2.2.8.1.7. Pragma Preserve_Layout

The TeleGen2 compiler reorders record components to minimize gaps within records. Pragma Preserve_Layout forces the compiler to maintain the Ada source order of components of a given record type, thereby preventing the compiler from performing this record layout optimization.

The syntax of this pragma is

```
Pragma Preserve_Layout ( ON => Record_Type_Name )
```

Preserve_Layout must appear before any forcing occurrences of the record type and must be in the same declarative part, package specification, or task specification. This pragma can be applied to a record type that has been packed. If Preserve_Layout is applied to a record type that has a record representation clause, the pragma only applies to the components that do not have component clauses. These components will appear in Ada source order after the components with component clauses.

2.2.8.1.8. Pragma Suppress_All

Suppress_All is a TeleGen2-defined pragma that will suppress all checks in a given scope. Pragma Suppress_All contains no arguments and can be placed in the same scopes as pragma Suppress.

In the absence of pragma Suppress_All or any other suppress pragma, the scope which contains the pragma will have checking turned off. This pragma should be used in a safe piece of time critical code to allow for better performance.

2.2.8.2. Implementation-dependent attributes

2.2.8.2.1. 'Address and 'Offset

These were discussed within the context of using machine code insertions, Section 2.1.4.2.2, "Implementation-dependent attributes to access Ada Objects."

2.2.8.2.2. Extended attributes for scalar types

The extended attributes extend the concept behind the Text_IO attributes 'Image, 'Value, and 'Width to give the user more power and flexibility when displaying values of scalars. Extended attributes differ in two respects from their predefined counterparts:

1. Extended attributes take more parameters and allow control of the format of the output string.

2. Extended attributes are defined for all scalar types, including fixed and floating point types.

Extended versions of predefined attributes are provided for integer, enumeration, floating point, and fixed point types:

Integer:	'Extended_Image,	'Extended_Value,	'Extended_Width
Enumeration:	'Extended_Image,	'Extended_Value,	'Extended_Width
Floating Point:	'Extended_Image,	'Extended_Value,	'Extended_Digits
Fixed Point:	'Extended_Image, 'Extended_Aft	'Extended_Value,	'Extended_Fore,

The extended attributes can be used without the overhead of including Text_IO in the linked program. The following are examples that illustrates the difference between instantiating Text_IO.Float_IO to convert a float value to a string and using Float'Extended_Image:

```
with Text_IO;
function Convert_To_String ( Fl : Float ) return String is
  Temp_Str : String ( 1 .. 6 + Float'Digits );
package Flt_IO is new Text_IO.Float_IO (Float);
begin
  Flt_IO.Put ( Temp_Str, Fl );
  return Temp_Str;
end Convert_To_String;



function Convert_To_String_No_Text_IO( Fl : Float ) return String is
begin
  return Float'Extended_Image ( Fl );
end Convert_To_String_No_Text_IO;



with Text_IO, Convert_To_String, Convert_To_String_No_Text_IO;
procedure Show_Different_Conversions is
  Value : Float := 10.03376;
begin
  Text_IO.Put_Line ( "Using the Convert_To_String, the value of
  the variable is : " & Convert_To_String ( Value ) );
  Text_IO.Put_Line ( "Using the Convert_To_String_No_Text_IO,
  the value is : " & Convert_To_String_No_Text_IO ( Value ) );
end Show_Different_Conversions;
```

2.2.8.2.2.1. Integer attributes

'Extended_Image

```
X'Extended_Image(Item,Width,Base,Based,Space_If_Positive)
```

Returns the image associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that the value of Item is an integer literal with no underlines, no exponent, no leading zeros (but a single zero for the zero value), and a minus sign if negative. If the resulting sequence of characters to be output has fewer than Width characters, leading spaces are first output to make up the difference. (LRM 14.3.7:10,14.3.7:11)

For a prefix X that is a discrete type or subtype, this attribute is a function that may have more than one parameter. The parameter Item must be an integer value. The resulting string is without underlines, leading zeros, or trailing spaces.

Parameters

`Item` The item for which you want the image; it is passed to the function. Required.

`Width` The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. Optional.

`Base` The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. Optional.

`Based` An indication of whether you want the string returned to be in base notation or not. If no preference is specified, the default (false) is assumed. Optional.

`Space_If_Positive` An indication of whether or not a positive integer should be prefixed with a space in the string returned. If no preference is specified, the default (false) is assumed. Optional.

Examples

```
subtype X is Integer Range -10..16;
```

Values yielded for selected parameters:

```
X'Extended_Image(5)                    = "5"
X'Extended_Image(5,0)                  = "5"
X'Extended_Image(5,2)                  = " 5"
X'Extended_Image(5,0,2)                = "101"
```

```
X'Extended_Image(5,4,2)                 = " 101"
X'Extended_Image(5,0,2,True)            = "2#101#"
X'Extended_Image(5,0,10,False)          = "5"
X'Extended_Image(5,0,10,False,True)     = " 5"
X'Extended_Image(-1,0,10,False,False)   = "-1"
X'Extended_Image(-1,0,10,False,True)    = "-1"
X'Extended_Image(-1,1,10,False,True)    = "-1"
X'Extended_Image(-1,0,2,True,True)      = "-2#1#"
X'Extended_Image(-1,10,2,True,True)     = "      -2#1#"
```

'Extended_Value

```
X'Extended_Value(Item)
```

Returns the value associated with Item as defined in Text_IO.Integer_IO. The Text_IO definition states that given a string, it reads an integer value from the beginning of the string. The value returned corresponds to the sequence input. (LRM 14.3.7:14)

For a prefix X that is a discrete type or subtype, this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter

`Item` A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type X. Required.

Examples

```
subtype X is Integer Range -10..16;
```

Values yielded for selected parameters:

```
X'Extended_Value("5")          = 5
X'Extended_Value(" 5")         = 5
X'Extended_Value("2#101#")     = 5
X'Extended_Value("-1")         = -1
X'Extended_Value(" -1")        = -1
```

'Extended_Width

```
X'Extended_Width(Base,Based,Space_If_Positive)
```

Returns the width for subtype of X. For a prefix X that is a discrete subtype, this attribute is a function that may have multiple parameters. This attribute yields the maximum image length over all values of the type or subtype X.

Parameters

`Base` The base for which the width will be calculated. If no base is specified, the default (10) is assumed. Optional.

`Based` An indication of whether the subtype is stated in based notation. If no value for based is specified, the default (false) is assumed. Optional.

`Space_If_Positive` An indication of whether or not the sign bit of a positive integer is included in the string returned. If no preference is specified, the default (false) is assumed. Optional.

Examples

```
subtype X is Integer Range -10..16;
```

Values yielded for selected parameters:

```
X'Extended_Width                    = 3  -- "-10"
X'Extended_Width(10)                = 3  -- "-10"
X'Extended_Width(2)                 = 5  -- "10000"
X'Extended_Width(10,True)           = 7  -- "-10#10#"
X'Extended_Width(2,True)            = 8  -- "2#10000#"
X'Extended_Width(10,False,True)     = 3  -- " 16"
X'Extended_Width(10,True,False)     = 7  -- "-10#10#"
X'Extended_Width(10,True,True)      = 7  -- " 10#16#"
X'Extended_Width(2,True,True)       = 9  -- " 2#10000#"
X'Extended_Width(2,False,True)      = 6  -- " 10000"
```

2.2.8.2.2.2. Enumeration type attributes

'Extended_Image

```
X'Extended_Image(Item,Width,Uppercase)
```

Returns the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that given an enumeration literal, it will output the value of the enumeration literal (either an identifier or a character literal). The character case parameter is ignored for character literals. (LRM 14.3.9:9)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more that one parameter. The parameter Item must be an enumeration value. The image of an enumeration value is the corresponding identifier, which may have character case and return string width specified.

Parameters

`Item` The item for which you want the image; it is passed to the function. Required.

`Width` The minimum number of characters to be in the string that is returned. If no width is specified, the default (0) is assumed. If the Width specified is larger than the image of Item, the return string is padded with trailing spaces. If the Width specified is smaller than the image of Item, the default is assumed and the image of the enumeration value is output completely. Optional.

`Uppercase` An indication of whether the returned string is in uppercase characters. In the case of an enumeration type where the enumeration literals are character literals, Uppercase is ignored and the case specified by the type definition is taken. If no preference is specified, the default (true) is assumed. Optional.

Examples

```
type X is (red, green, blue, purple);
type Y is ('a', 'B', 'c', 'D');
```

Values yielded for selected parameters:

```
X'Extended_Image(red)              = "RED"
X'Extended_Image(red, 4)           = "RED "
X'Extended_Image(red,2)            = "RED"
X'Extended_Image(red,0,false)      = "red"
X'Extended_Image(red,10,false)     = "red       "
Y'Extended_Image('a')              = "'a'"
Y'Extended_Image('B')              = "'B'"
Y'Extended_Image('a',6)            = "'a'  "
Y'Extended_Image('a',0,true)       = "'a'"
```

'Extended_Value

```
X'Extended_Value(Item)
```

Returns the image associated with Item as defined in Text_IO.Enumeration_IO. The Text_IO definition states that it reads an enumeration value from the beginning of the given string and returns the value of the enumeration literal that corresponds to the sequence input. (LRM 14.3.9:11)

For a prefix X that is a discrete type or subtype, this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter

`Item` A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of X. Required.

Examples

```
type X is (red, green, blue, purple);
```

Values yielded for selected parameters:

```
X'Extended_Value("red")          = red
X'Extended_Value(" green")       = green
X'Extended_Value("    Purple")   = purple
X'Extended_Value(" GreEn  ")     = green
```

'Extended_Width

```
X'Extended_Width
```

Returns the width for subtype of X.

For a prefix X that is a discrete type or subtype; this attribute is a function. This attribute yields the maximum image length over all values of the enumeration type or subtype X.

Parameters

There are no parameters to this function. This function returns the width of the largest (width) enumeration literal in the enumeration type specified by X.

Examples

```
type X is (red, green, blue, purple);
type Z is (X1, X12, X123, X1234);
```

Values yielded:

```
X'Extended_Width      = 6  -- "purple"
Z'Extended_Width      = 5  -- "X1234"
```

2.2.8.2.2.3. Floating point attributes

'Extended_Image

```
X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)
```

Returns the image associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameters

`Item` The item for which you want the image; it is passed to the function. Required.

`Fore` The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified, the default value (2) is assumed. Optional.

`Aft` The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. Optional.

`Exp` The minimum number of digits in the exponent. The exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. Optional.

`Base` The base that the image is to be displayed in. If no base is specified, the default (10) is assumed. Optional.

`Based` An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. Optional.

Examples

```
type X is digits 5 range -10.0 .. 16.0;
```

Values yielded for selected parameters:

```
X'Extended_Image(5.0)                    = " 5.0000E+00"
X'Extended_Image(5.0,1)                  = "5.0000E+00"
X'Extended_Image(-5.0,1)                 = "-5.0000E+00"
X'Extended_Image(5.0,2,0)                = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)              = " 5.0"
X'Extended_Image(5.0,2,0,0,2)            = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)       = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)       = "2#1.1#E+02"
```

'Extended_Value

```
X'Extended_Value(Item)
```

Returns the value associated with Item as defined in Text_IO.Float_IO. The Text_IO definition states that it skips any leading zeros, then reads a plus or minus sign if present then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter

`Item` A parameter of the predefined type string; it is passed to the function. The type of the returned value is the base type of the input string. Required.

Examples

```
type X is digits 5 range -10.0 .. 16.0;
```

Values yielded for selected parameters:

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
X'Extended_Value("2#1.01#E2")    = 5.0
```

'Extended_Digits

```
X'Extended_Digits(Base)
```

Returns the number of digits using base in the mantissa of model numbers of the subtype X.

Parameter

`Base`	The base that the subtype is defined in. If no base is specified, the default (10) is assumed. Optional.

Examples

```
type X is digits 5 range -10.0 .. 16.0;
```

Values yielded:

```
X'Extended_Digits     = 5
```

2.2.8.2.2.4. Fixed-point attributes

'Extended_Image

```
X'Extended_Image(Item,Fore,Aft,Exp,Base,Based)
```

Returns the image associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it outputs the value of the parameter Item as a decimal literal with the format defined by the other parameters. If the value is negative, a minus sign is included in the integer part of the value of Item. If Exp is 0, the integer part of the output has as many digits as are needed to represent the integer part of the value of Item or is zero if the value of Item has no integer part. (LRM 14.3.8:13, 14.3.8:15)

For a prefix X that is a discrete type or subtype; this attribute is a function that may have more than one parameter. The parameter Item must be a Real value. The resulting string is without underlines or trailing spaces.

Parameters

`Item` The item for which you want the image; it is passed to the function. Required.

`Fore` The minimum number of characters for the integer part of the decimal representation in the return string. This includes a minus sign if the value is negative and the base with the '#' if based notation is specified. If the integer part to be output has fewer characters than specified by Fore, leading spaces are output first to make up the difference. If no Fore is specified, the default value (2) is assumed. Optional.

`Aft` The minimum number of decimal digits after the decimal point to accommodate the precision desired. If the delta of the type or subtype is greater than 0.1, then Aft is 1. If no Aft is specified, the default (X'Digits-1) is assumed. If based notation is specified, the trailing '#' is included in Aft. Optional.

`Exp` The minimum number of digits in the exponent; the exponent consists of a sign and the exponent, possibly with leading zeros. If no Exp is specified, the default (3) is assumed. If Exp is 0, no exponent is used. Optional.

`Base` The base in which the image is to be displayed. If no base is specified, the default (10) is assumed. Optional.

`Based` An indication of whether you want the string returned to be in based notation or not. If no preference is specified,

the default (false) is assumed. Optional.

Examples

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Values yielded for selected parameters:

```
X'Extended_Image(5.0)                    = " 5.00E+00"
X'Extended_Image(5.0,1)                  = "5.00E+00"
X'Extended_Image(-5.0,1)                 = "-5.00E+00"
X'Extended_Image(5.0,2,0)                = " 5.0E+00"
X'Extended_Image(5.0,2,0,0)              = " 5.0"
X'Extended_Image(5.0,2,0,0,2)            = "101.0"
X'Extended_Image(5.0,2,0,0,2,True)       = "2#101.0#"
X'Extended_Image(5.0,2,2,3,2,True)       = "2#1.1#E+02"
```

'Extended_Value

```
X'Extended_Value(Image)
```

Returns the value associated with Item as defined in Text_IO.Fixed_IO. The Text_IO definition states that it skips any leading zeros, reads a plus or minus sign if present, then reads the string according to the syntax of a real literal. The return value is that which corresponds to the sequence input. (LRM 14.3.8:9, 14.3.8:10)

For a prefix X that is a discrete type or subtype; this attribute is a function with a single parameter. The actual parameter Item must be of predefined type string. Any leading or trailing spaces in the string X are ignored. In the case where an illegal string is passed, a Constraint_Error is raised.

Parameter

`Image`	Parameter of the predefined type string. The type of the returned value is the base type of the input string. Required.

Examples

```
type X is delta 0.1 range -10.0 .. 17.0;
```

Values yielded for selected parameters:

```
X'Extended_Value("5.0")          = 5.0
X'Extended_Value("0.5E1")        = 5.0
X'Extended_Value("2#1.01#E2")    = 5.0
```

'Extended_Fore

```
X'Extended_Fore(Base,Based)
```

Returns the minimum number of characters required for the integer part of the based representation of X.

Parameters

`Base`	The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. Optional.
`Based`	An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. Optional.

Examples

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Values yielded for selected parameters:

```
X'Extended_Fore     = 3  -- "-10"
X'Extended_Fore(2)  = 6  -- " 10001"
```

'Extended_Aft

`X'Extended_Aft(Base,Based)`

Returns the minimum number of characters required for the fractional part of the based representation of X.

Parameters

`Base` The base in which the subtype is to be displayed. If no base is specified, the default (10) is assumed. Optional.

`Based` An indication of whether you want the string returned to be in based notation or not. If no preference is specified, the default (false) is assumed. Optional.

Examples

```
type X is delta 0.1 range -10.0 .. 17.1;
```

Values yielded for selected parameters:

```
X'Extended_Aft      = 1   - "1" from 0.1
X'Extended_Aft(2)   = 4   - "0001" from 2#0.0001#
```

2.2.8.3. Package System

```
with Unchecked_Conversion;

package System is

   --=====================
   -- CUSTOMIZABLE VALUES
   --=====================

   type Name     is (TeleGen2);

   System_Name   : constant name := TeleGen2;

   Memory_Size   : constant := (2 ** 31) -1; --Available memory, in storage units
   Tick          : constant := 1.0 / 50.0;   --Basic clock rate, in seconds

   type Task_Data is    --
     record             -- Adaptation-specific customization information
       null;            -- for task objects.
     end record;        --

   --=============================================================
   -- NON-CUSTOMIZABLE, IMPLEMENTATION-DEPENDENT VALUES
   --=============================================================

   Storage_Unit : constant := 8;
   Min_Int      : constant := -(2 ** 31);
   Max_Int      : constant := (2 ** 31) - 1;
   Max_Digits   : constant := 15;
   Max_Mantissa : constant := 31;
   Fine_Delta   : constant := 1.0 / (2 ** Max_Mantissa);

   subtype Priority is Integer Range 0 .. 63;

   --=======================
   -- ADDRESS TYPE SUPPORT
   --=======================

   type Memory is private;
   type Address is access Memory;
     --
     -- Ensures compatibility between addresses and access types.
     -- Also provides implicit NULL initial value.

   Null_Address: constant Address := null;
     --
     -- Initial value for any Address object

   type Address_Value is range -(2**31)..(2**31)-1;
     --
     -- A numeric representation of logical addresses for use in address clauses
```

```
Hex_80000000 : constant Address_Value := - 16#80000000#;
Hex_90000000 : constant Address_Value := - 16#70000000#;
Hex_A0000000 : constant Address_Value := - 16#60000000#;
Hex_B0000000 : constant Address_Value := - 16#50000000#;
Hex_C0000000 : constant Address_Value := - 16#40000000#;
Hex_D0000000 : constant Address_Value := - 16#30000000#;
Hex_E0000000 : constant Address_Value := - 16#20000000#;
Hex_F0000000 : constant Address_Value := - 16#10000000#;
  --
  -- Define numeric offsets to aid in Address calculations
  -- Example:
  --   for Hardware use at Location (Hex_F0000000 + 16#2345678#);

function Location is new Unchecked_Conversion (Address_Value, Address);
  --
  -- May be used in address clauses:
  --
  --   Object: Some_Type;
  --   for Object use at Location (16#4000#);

function Label (Name: String) return Address;
pragma Interface (META, Label);
  --
  -- The LABEL meta-function allows a link name to be specified as address
  -- for an imported object in an address clause:
  --
  --  Object: Some_Type;
  --  for Object use at Label("OBJECT$$LINK_NAME");
  --
  -- System.Label returns Null_Address for non-literal parameters.

--=========================
-- ERROR REPORTING SUPPORT
--=========================

procedure Report_Error;
pragma Interface (Assembly, Report_Error);
pragma Interface_Information (Report_Error, "_REPORT_ERROR");

--
-- Report_Error can only be called as the very first thing in an exception
--   handler (all other uses are erroneous) and provides
--   an exception traceback like tracebacks provided for unhandled
--   exceptions
--

--============
-- CALL SUPPORT
--============

 type Subprogram_Value IS
 record
    Proc_addr     : Address;
```

```
      Parent_frame : Address;
   end record;

    --
    -- Value returned by the implementation-defined 'Subprogram_Value
    -- attribute. The attribute is not defined for subprograms with
    -- parameters, or functions.

  Max_Object_Size   : CONSTANT := Max_Int;
  Max_Record_Count  : CONSTANT := Max_Int;
  Max_Text_Io_Count : CONSTANT := Max_Int-1;
  Max_Text_Io_Field : CONSTANT := 1000;

private
   type Memory is
   record
     null;
   end record;

end System;
```

2.2.8.3.1. System.Label

The System.Label meta-function is provided to allow users to address objects by a linker-recognized label name. This function takes a single string literal as a parameter and returns a value of System.Address. The function simply returns the run-time address of the appropriate resolved link name, the primary purpose being to address objects created and referenced from other languages.

- When used in an address clause, System.Label indicates that the Ada object or subprogram is to be referenced by a label name. The actual object must be created in some other unit (normally by another language), and this capability simply allows the user to import that object and reference it in Ada.

- When used in an expression, System.Label provides the link time address of any name; a name that might be for an object, a subprogram, etc.

2.2.8.3.2. System.Report_Error

Report_Error must only be called from within an exception handler, and must be the first thing done within it. This routine displays the normal exception traceback information to standard output. It is essentially the same traceback that could be obtained if the exception were unhandled and propagated out of the program, but the user may want to handle the exception and still display this information. The user may also want to use this capability in a user handler at the end of a task (since those exceptions will not be propagated to the main program). Note that the user can also get this capability for all tasks using the -X binder switch.

For details on the output, refer to Section 2.1.2, "Exceptions."